MORGAN & CLAYPOOL PUBLISHERS

Microchip (ATMEL) AVR® Microcontroller Primer

Programming and Interfacing
Third Edition

Steven F. Barrett, Daniel J. Pack

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS

Mitchell A. Thornton, *Series Editor*

Microchip AVR® Microcontroller Primer: Programming and Interfacing

Third Edition

Synthesis Lectures on Digital Circuits and Systems

The *Synthesis Lectures on Digital Circuits and Systems* series is comprised of 50- to 100-page books targeted for audience members with a wide-ranging background. The Lectures include topics that are of interest to students, professionals, and researchers in the area of design and analysis of digital circuits and systems. Each Lecture is self-contained and focuses on the background information required to understand the subject matter and practical case studies that illustrate applications. The format of a Lecture is structured such that each will be devoted to a specific topic in digital circuits and systems rather than a larger overview of several topics such as that found in a comprehensive handbook. The Lectures cover both well-established areas as well as newly developed or emerging material in digital circuits and systems design and analysis.

Microchip AVR® Microcontroller Primer: Programming and Interfacing, Third Edition
Steven F. Barrett and Daniel J. Pack
2019

Synthesis of Quantum Circuits vs. Synthesis of Classical Reversible Circuits
Alexis De Vos, Stijn De Baerdemacker, and Yvan Van Rentergen
2018

Boolean Differential Calculus
Bernd Steinbach and Christian Posthoff
2017

Embedded Systems Design with Texas Instruments MSP432 32-bit Processor
Dung Dang, Daniel J. Pack, and Steven F. Barrett
2016

Fundamentals of Electronics: Book 4 Oscillators and Advanced Electronics Topics
Thomas F. Schubert and Ernest M. Kim
2016

Fundamentals of Electronics: Book 3 Active Filters and Amplifier Frequency
Thomas F. Schubert and Ernest M. Kim
2016

Bad to the Bone: Crafting Electronic Systems with BeagleBone and BeagleBone Black, Second Edition
Steven F. Barrett and Jason Kridner
2015

Fundamentals of Electronics: Book 2 Amplifiers: Analysis and Design
Thomas F. Schubert and Ernest M. Kim
2015

Fundamentals of Electronics: Book 1 Electronic Devices and Circuit Applications
Thomas F. Schubert and Ernest M. Kim
2015

Applications of Zero-Suppressed Decision Diagrams
Tsutomu Sasao and Jon T. Butler
2014

Modeling Digital Switching Circuits with Linear Algebra
Mitchell A. Thornton
2014

Arduino Microcontroller Processing for Everyone! Third Edition
Steven F. Barrett
2013

Boolean Differential Equations
Bernd Steinbach and Christian Posthoff
2013

Bad to the Bone: Crafting Electronic Systems with BeagleBone and BeagleBone Black
Steven F. Barrett and Jason Kridner
2013

Introduction to Noise-Resilient Computing
S.N. Yanushkevich, S. Kasai, G. Tangim, A.H. Tran, T. Mohamed, and V.P. Shmerko
2013

Atmel AVR Microcontroller Primer: Programming and Interfacing, Second Edition
Steven F. Barrett and Daniel J. Pack
2012

Representation of Multiple-Valued Logic Functions
Radomir S. Stankovic, Jaakko T. Astola, and Claudio Moraga
2012

Arduino Microcontroller: Processing for Everyone! Second Edition
Steven F. Barrett
2012

Advanced Circuit Simulation Using Multisim Workbench
David Báez-López, Félix E. Guerrero-Castro, and Ofelia Delfina Cervantes-Villagómez
2012

Circuit Analysis with Multisim
David Báez-López and Félix E. Guerrero-Castro
2011

Microcontroller Programming and Interfacing Texas Instruments MSP430, Part I
Steven F. Barrett and Daniel J. Pack
2011

Microcontroller Programming and Interfacing Texas Instruments MSP430, Part II
Steven F. Barrett and Daniel J. Pack
2011

Pragmatic Electrical Engineering: Systems and Instruments
William Eccles
2011

Pragmatic Electrical Engineering: Fundamentals
William Eccles
2011

Introduction to Embedded Systems: Using ANSI C and the Arduino Development Environment
David J. Russell
2010

Arduino Microcontroller: Processing for Everyone! Part II
Steven F. Barrett
2010

Arduino Microcontroller Processing for Everyone! Part I
Steven F. Barrett
2010

Digital System Verification: A Combined Formal Methods and Simulation Framework
Lun Li and Mitchell A. Thornton
2010

Progress in Applications of Boolean Functions
Tsutomu Sasao and Jon T. Butler
2009

Embedded Systems Design with the Atmel AVR Microcontroller: Part II
Steven F. Barrett
2009

Embedded Systems Design with the Atmel AVR Microcontroller: Part I
Steven F. Barrett
2009

Embedded Systems Interfacing for Engineers using the Freescale HCS08 Microcontroller II: Digital and Analog Hardware Interfacing
Douglas H. Summerville
2009

Designing Asynchronous Circuits using NULL Convention Logic (NCL)
Scott C. Smith and JiaDi
2009

Embedded Systems Interfacing for Engineers using the Freescale HCS08 Microcontroller I: Assembly Language Programming
Douglas H.Summerville
2009

Developing Embedded Software using DaVinci & OMAP Technology
B.I. (Raj) Pawate
2009

Mismatch and Noise in Modern IC Processes
Andrew Marshall
2009

Asynchronous Sequential Machine Design and Analysis: A Comprehensive Development of the Design and Analysis of Clock-Independent State Machines and Systems
Richard F. Tinder
2009

An Introduction to Logic Circuit Testing
Parag K. Lala
2008

Pragmatic Power
William J. Eccles
2008

Multiple Valued Logic: Concepts and Representations
D. Michael Miller and Mitchell A. Thornton
2007

Finite State Machine Datapath Design, Optimization, and Implementation
Justin Davis and Robert Reese
2007

Atmel AVR Microcontroller Primer: Programming and Interfacing
Steven F. Barrett and Daniel J. Pack
2007

Pragmatic Logic
William J. Eccles
2007

PSpice for Filters and Transmission Lines
Paul Tobin
2007

PSpice for Digital Signal Processing
Paul Tobin
2007

PSpice for Analog Communications Engineering
Paul Tobin
2007

PSpice for Digital Communications Engineering
Paul Tobin
2007

PSpice for Circuit Theory and Electronic Devices
Paul Tobin
2007

Pragmatic Circuits: DC and Time Domain
William J. Eccles
2006

Pragmatic Circuits: Frequency Domain
William J. Eccles
2006

Pragmatic Circuits: Signals and Filters
William J. Eccles
2006

High-Speed Digital System Design
Justin Davis
2006

Introduction to Logic Synthesis using Verilog HDL
Robert B.Reese and Mitchell A.Thornton
2006

Microcontrollers Fundamentals for Engineers and Scientists
Steven F. Barrett and Daniel J. Pack
2006

Microchip AVR® Microcontroller Primer:
Programming and Interfacing, Third Edition
Steven F. Barrett and Daniel J. Pack
www.morganclaypool.com

ISBN: 9781681732046 paperback
ISBN: 9781681732053 ebook
ISBN: 9781681736235 hardcover

DOI 10.2200/S00803ED3V01Y201709DCS053

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS

Lecture #53
Series Editor: Mitchell A. Thornton, *Southern Methodist University*
Series ISSN
Print 1932-3166 Electronic 1932-3174

Microchip AVR® Microcontroller Primer: Programming and Interfacing

Third Edition

Steven F. Barrett
University of Wyoming, Laramie, WY

Daniel J. Pack
University of Tennessee Chattanooga, TN

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS #53

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

This textbook provides practicing scientists and engineers a primer on the Microchip AVR® microcontroller. The revised title of this book reflects the 2016 Microchip Technology acquisition of Atmel Corporation. In this third edition we highlight the popular ATmega164 microcontroller and other pin-for-pin controllers in the family with a complement of flash memory up to 128 KB. The third edition also provides an update on Atmel Studio, programming with a USB pod, the gcc compiler, the ImageCraft JumpStart C for AVR compiler, the Two-Wire Interface (TWI), and multiple examples at both the subsystem and system level. Our approach is to provide readers with the fundamental skills to quickly set up and operate with this internationally popular microcontroller. We cover the main subsystems aboard the ATmega164, providing a short theory section followed by a description of the related microcontroller subsystem with accompanying hardware and software to operate the subsystem. In all examples, we use the C programming language. We include a detailed chapter describing how to interface the microcontroller to a wide variety of input and output devices and conclude with several system level examples including a special effects light-emitting diode cube, autonomous robots, a multi-function weather station, and a motor speed control system.

KEYWORDS

microchip microcontroller, Microchip AVR®, ATmega164, microcontroller interfacing, embedded systems design

To our families

Contents

Preface

In 2006, Morgan & Claypool Publishers (M&C) released our textbook *Microcontrollers Fundamentals for Engineers and Scientists*. The purpose of this textbook was to provide practicing scientists and engineers a tutorial on the fundamental concepts and the use of microcontrollers. The textbook presented the fundamental concepts common to all microcontrollers. Our goal for writing this follow–on book is to present details on a specific microcontroller family, the Microchip AVR® ATmega164 Microcontroller family. This family includes the ATmega164 microcontroller which is equipped with four 8-bit input/output ports, a plethora of subsystems, and a 16K–bytes flash program memory. Other microcontrollers in the family include the pin–for–pin compatible ATmega324 (32K–bytes program memory), ATmega644 (64K–bytes program memory), and the ATmega1284 (128K–bytes program memory).

Why Microchip microcontrollers? There are many excellent international companies that produce microcontrollers. Some of the highlights of the Microchip AVR® microcontroller line include:

- high performance coupled with low power consumption,
- outstanding flash memory technology,
- reduced instruction set computer Harvard Architecture,
- single-cycle instruction execution,
- wide variety of operating voltages (1.8–5.5 VDC),
- architecture designed for the C language,
- one set of development tools for the entire AVR® microcontroller line, and
- in–system programming, debugging, and verification capability.

Although all of these features are extremely important, we have chosen to focus on the Microchip AVR® microcontroller line of microcontrollers for this primer for a number of other related reasons.

- The learning curve for Microchip microcontrollers is gentle. If this is your first exposure to microcontrollers, you will quickly come up to speed on microcontroller programming and interfacing. If you already know another line of processors, you can quickly apply your knowledge to this powerful line of 8-bit processors.

- It is relatively inexpensive to get started with the Microchip AVR® microcontroller line. The microcontrollers themselves are inexpensive, and the compilers and programming hardware and software are easily affordable.

- The AVR® microcontroller line provides a full range of processing power, from small, yet powerful 8-pin processors to complex 100-pin processors. The same compiler and programming hardware may be used with a wide variety of microcontrollers.

- Many of the AVR® microcontrollers are available in dual inline package (DIP), which makes them readily useable on a printed circuit board prototype (e.g., capstone design projects).

- Many of the microcontrollers in the AVR® microcontroller line are pin-for-pin compatible with one another. This allows you to easily move up and down the AVR® microcontroller line as your project becomes better defined.

- Microchip has documentation available for every microcontroller at your fingertips. Simply visit `www.microchip.com`. Furthermore, Microchip customer support is good and responsive.

- There is worldwide interest in the AVR® microcontroller line. We would be remiss to not mention AVR Freaks®. This is a dedicated, international group of AVR® microcontroller experts who share their expertise online with other high–power users and novices alike.

Approach of the book

If this is your first exposure to microcontrollers, we highly recommend that you read first our M&C textbook, *Microcontrollers Fundamentals for Engineers and Scientists*. It will provide you the background information necessary to fully appreciate the contents of this textbook. This textbook picks up where the first one left off. We have received permission from M&C to include some of the background material from the first textbook in this text to allow for a complete stand-alone product.

Our approach in this textbook is to provide you with the fundamental skills to quickly get set up and operate an Microchip microcontroller. We have chosen to use the AVR® ATmega164 as a representative sample of the AVR® microcontroller line (more on this processor later). The knowledge you gain on the ATmega164 can be easily translated to every other microcontroller in the AVR® microcontroller line.

The M&C textbooks are designed to be short tutorials on a given topic. Therefore, our treatment of each topic will provide a short theory section followed by a description of the related microcontroller subsystem with accompanying hardware and software to exercise the subsystem. In all examples, we use the C programming language. There are many excellent C compilers available for the Microchip AVR® microcontoller line. We have chosen the Atmel Studio integrated development platform (IDP) for its short learning curve and ease of use. We also provide examples using the ImageCraft JumpStart C for AVR compiler `www.imagecraft.com`. We use the USB compatible Microchip AVR Dragon employing In–System Programming (ISP) techniques.

NEW IN THE THIRD EDITION

The third edition provides the following updated and expanded features:

- examples provided for the ATmega164. Easily ported to other compatible ATmega processors;
- multiple new, fully worked examples;
- ISP programming with the Atmel AVR Dragon;
- examples using serial UART configured LCD, voice synthesis chip, and serial monitor on host PC;
- tri–color light-emitting diode (LED) strip controlled by SPI system;
- detailed example using TWI (I2C) system;
- detailed heart beat monitor example using input capture system;
- controlling AC load using PowerSwitch Tail II;
- extended examples for 3D special effects LED cube, maze robots, a weather station, motor speed control circuit; and
- support provided for both the Atmel AVR Studio gcc compiler and the ImageCraft JumpStart C for AVR compiler.

Steven F. Barrett and Daniel J. Pack
Laramie, WY and Chattanooga, TN
September 2019

Acknowledgments

There have been many people involved in the conception and production of this book. In 2005, Joel Claypool of Morgan & Claypool Publishers, invited us to write a book on microcontrollers for his new series titled "Synthesis Lectures on Digital Circuits and Systems." The result was the book *Microcontrollers Fundamentals for Engineers and Scientists.* Since then we have been regular contributors to the series. Our goal has been to provide the fundamental concepts common to all microcontrollers and then apply the concepts to the specific microcontroller under discussion. We believe that once you have mastered these fundamental concepts, they are easily transportable to different processors. As with many other projects, he has provided his publishing expertise to convert our final draft into a finished product. We thank him for his support on this project and many others. He has provided many novice writers the opportunity to become published authors. His vision and expertise in the publishing world made this book possible. We thank Sara Kreisman of Rambling Rose Press, Inc. for her editorial expertise. We also thank Dr. C.L. Tondo of T&T TechWorks, Inc. and his staff for working their magic to convert our final draft into a beautiful book.

We also thank Sparkfun, Adafruit, ImageCraft, and Microchip for their permission to use their copyrighted material and screenshots throughout the text. Several Microchip acknowledgments are in order.

- This book contains copyrighted material of Microchip Technology Incorporated replicated with permission. All rights reserved. No further replications may be made without Microchip Technology Inc's prior written consent.

- *Microchip AVR® Microcontroller Primer: Programming and Interfacing, Third Edition* is an independent publication and is not affiliated with, nor has it been authorized, sponsored, or otherwise approved by Microchip.

Most of all, we thank our families. Our work could not have come to fruition without the sacrifice and encouragement of our families over the past fifteen plus years. Without you, none of this would matter. We love you!

Steven F. Barrett and Daniel J. Pack
Laramie, WY and Chattanooga, TN
September 2019

CHAPTER 1

Microchip AVR® Architecture Overview

Objectives: After reading this chapter, the reader should be able to:

- provide an overview of the RISC architecture of the ATmega164;
- describe the different ATmega164 memory components and their applications;
- explain the ATmega164 internal subsystems and their applications;
- highlight the operating parameters of the ATmega164; and
- summarize the special ATmega164 features.

1.1 ATMEGA164 ARCHITECTURE OVERVIEW

In this section, we describe the overall architecture of the Microchip AVR ATmega164. We begin with an introduction to the concept of the reduced instruction set computer (RISC) and briefly describe the Microchip Assembly Language Instruction Set. We program mainly in C throughout the course of the book. We then provide a detailed description of the ATmega164 hardware architecture.

1.1.1 REDUCED INSTRUCTION SET COMPUTER

In our first Morgan & Claypool (M&C) textbook [Barrett and Pack, 2006], we described a microcontroller as an entire computer system contained within a single integrated circuit or chip. Microcontroller operation is controlled by a user-written program interacting with the fixed hardware architecture resident within the microcontroller. A specific microcontroller architecture can be categorized as accumulator-based, register-based, stack-based, or a pipeline architecture.

The Microchip ATmega164 is a register-based architecture. In this type of architecture, both operands of an operation are stored in registers collocated with the central processing unit (CPU). This means that before an operation is performed, the computer loads all necessary data for the operation to its CPU. The result of the operation is also stored in a register. During program execution, the CPU interacts with the register set and minimizes slower memory accesses for both instructions and data. Memory accesses are typically handled as background operations.

Coupled with the register-based architecture is an instruction set based on the RISC concept. A RISC processor is equipped with a complement of very simple and efficient basic operations. More complex instructions are built up from these basic operations. This allows for efficient program operation. The Microchip ATmega164 is equipped with 131 RISC-type instructions. Most can be executed in a single clock cycle. The ATmega164 is also equipped with additional hardware to allow for the multiplication operation in two clock cycles. In many other microcontroller architectures, multiplication typically requires many more clock cycles. For additional information on the RISC architecture, the interested reader is referred to Hennessy and Patterson [2003].

The Microchip ATmega164 is equipped with 32 general-purpose 8-bit registers that are tightly coupled to the processor's arithmetic logic unit within the CPU. Also, the processor is designed following the Harvard Architecture format. That is, it is equipped with separate, dedicated memories and buses for program instructions and data information. The register-based Harvard Architecture coupled with the RISC-based instruction set allows for fast and efficient program execution and allows the processor to complete an assembly language instruction every clock cycle. Microchip indicates the ATmega164 can execute 20 million instructions per second when operating at a clock speed of 20 MHz and with the supply voltage between 4.5 and 5.5 VDC.

1.1.2 ASSEMBLY LANGUAGE INSTRUCTION SET

An instruction set is a group of instructions a machine "understands" how to execute. A large number of instructions provide flexibility but require more complex hardware. Thus, an instruction set is unique for a given hardware configuration and cannot be used with another hardware configuration. Microchip has equipped the ATmega164 with 131 different instructions.

For the most efficient and fast execution of a given microcontroller, assembly language should be used. Assembly language instructions are written to efficiently interact with a specific microcontroller's resident hardware. To effectively use the assembly language, the programmer must be thoroughly familiar with the low-level architecture details of the controller. Furthermore, the learning curve for a given assembly language is quite steep and lessons learned do not always transfer to another microcontroller. This is part of the reason programmers prefer to use a high-level language over an assembly language.

We program the Microchip ATmega164 using the C language throughout the text. The C programming language allows for direct control of microcontroller hardware at the register level while being portable to other microcontrollers in the AVR® microcontroller line. When a C program is compiled during the software development process, the program is first converted to assembly language and then to the machine code for the specific microcontroller.

We must emphasize that programming in C is not better than assembly language or vice versa. Both approaches have their inherent advantages and disadvantages. We have chosen to use C in this textbook for the reasons previously discussed.

1.1.3 ATMEGA164 ARCHITECTURE OVERVIEW

We have chosen the ATmega164 as a representative of the Microchip AVR line of microcontrollers. Lessons learned with the ATmega164 may be easily adapted to all other processors in the AVR microcontroller line. A block diagram of the Microchip ATmega164's architecture is provided in Figure 1.1.

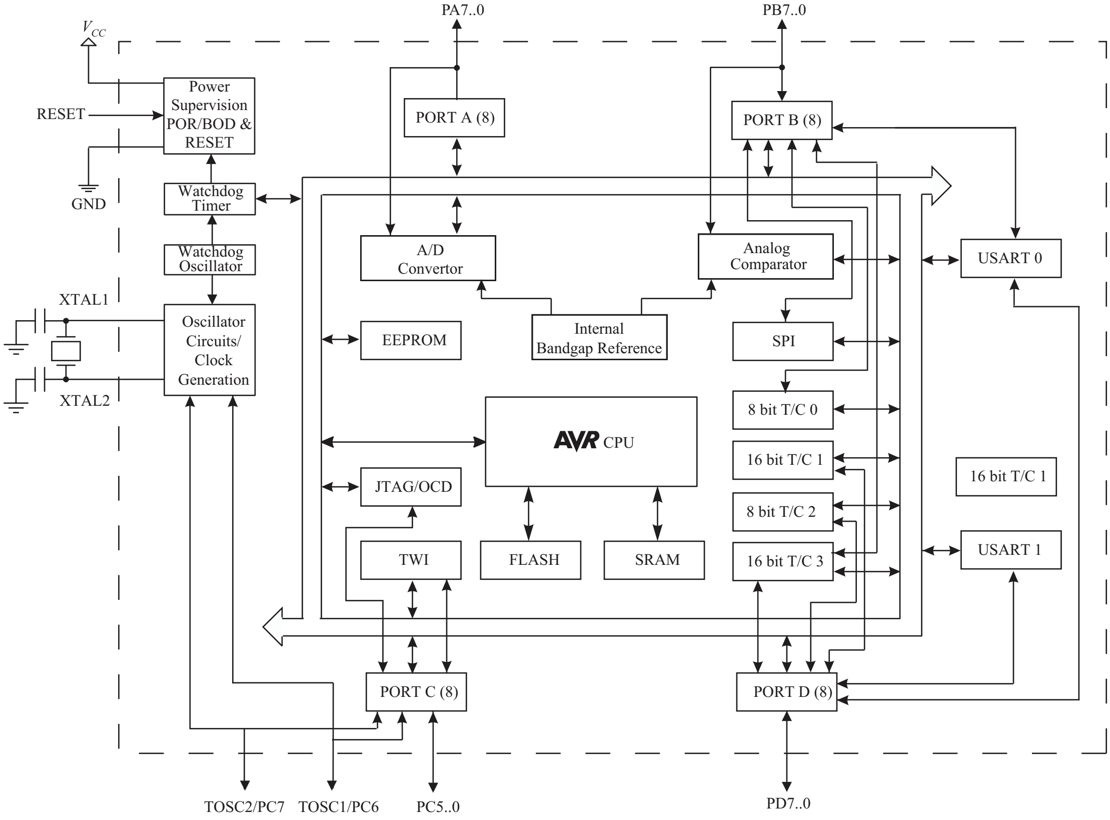


Figure 1.1: Microchip AVR ATmega164 block diagram. Figure used with permission of Microchip. All rights reserved.

As can be seen from the figure, the ATmega164 has external connections for power supplies (VCC, GND, AVCC, and AREF), an external time base (XTAL1 and XTAL2) input pins to drive its clocks, processor reset (active low RESET), and four 8-bit ports (PA0-PA7, PC0-PC7, PB0-PB7, and PD0-PD7), which are used to interact with the external world. These ports may be used as general purpose digital input/output (I/O) ports or they may be used for their alternate functions. The ports are interconnected with the ATmega164's CPU and internal subsystems via an internal bus. The ATmega164 also contains a timer subsystem, an analog-to-digital converter (ADC), an interrupt subsystem, memory components, and a serial communication subsystem.

In the next several sections, we briefly describe each of these internal subsystems shown in the figure. Detailed descriptions of selected subsystem operation and programming appear in latter parts of the book. Since we cannot cover all features of the microcontroller due to limited space, we focus on the primary functional components of the microcontroller to fulfill the purpose of this book as a basic primer to the ATmega164.

1.2 NONVOLATILE AND DATA MEMORIES

The ATmega164 is equipped with three main memory sections: flash electrically erasable programmable read-only memory (EEPROM), static random access memory (SRAM), and byte-addressable EEPROM for data storage. We discuss each memory component in turn.

1.2.1 IN-SYSTEM PROGRAMMABLE FLASH EEPROM

Bulk programmable flash EEPROM is typically used to store programs. It can be erased and programmed as a single unit. Also, should a program require a large table of constants, it may be included as a global variable within a program and programmed into flash EEPROM with the rest of the program. Flash EEPROM is nonvolatile, meaning memory contents are retained when microcontroller power is lost. The ATmega164 is equipped with 16 KB of onboard reprogrammable flash memory. This memory component is organized into 8 K locations, with 16 bits at each location.

The flash EEPROM is in-system programmable. In-system programmability (ISP) means the microcontroller can be programmed while resident within a circuit. It does not have to be removed from the circuit for programming. Instead, a host personal computer (PC), connected via a specialized programming cable and pod, downloads the program to the microcontroller. Alternately, the microcontroller can be programmed outside its resident circuit using a flash programmer board. We employ the AVR Dragon for ISP programming the ATmega164. This development board is readily available from a number of suppliers. It may be used to program many different microcontrollers in the ATmega and ATtiny product families.

1.2.2 BYTE-ADDRESSABLE EEPROM

Byte-addressable memory is used to permanently store and recall variables during program execution. It too is nonvolatile. That is, it retains its contents even when power is not available. It is especially useful for logging system malfunctions and fault data during program execution. It is also useful for storing data that must be retained during a power failure but might need to be changed periodically. Examples where this type of memory is used are found in applications to store system parameters, electronic lock combinations, and automatic garage door electronic unlock sequences. The ATmega164 is equipped with 512 B of byte-addressable EEPROM.

1.2.3 STATIC RANDOM ACCESS MEMORY

SRAM is volatile. That is, if the microcontroller loses power, the contents of SRAM memory are lost. It can be written to and read from during program execution. The ATmega164 is equipped with 1000 B (actually 1120) of SRAM. A small portion (96 locations) of the SRAM is set aside for the general-purpose registers used by the central processing unit (CPU) and also for the input/output (I/O) and peripheral subsystems aboard the microcontroller. A complete ATmega164 register listing and accompanying header file is provided in the Appendices. During program execution, RAM is used to store global variables, support dynamic memory allocation of variables, and provide a location for the stack (to be discussed later).

1.2.4 PROGRAMMABLE LOCK BITS

To provide for memory security from tampering, the ATmega164 is equipped with memory lock bits. These lock bits are programmed using the Microchip AVR Dragon. The lock bits may be configured for the following options:

- no memory lock features enabled;
- no further programming of memory is allowed using parallel or serial programming techniques; or
- no further programming or verification of memory is allowed using parallel or serial programming techniques.

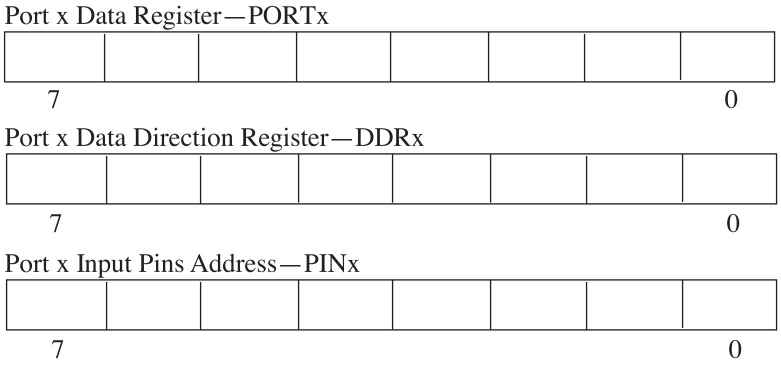
1.3 PORT SYSTEM

The Microchip ATmega164 is equipped with four 8-bit general-purpose, digital I/O ports designated PORTA, PORTB, PORTC, and PORTD. All of these ports also have alternate functions, which are described later. In this section, we concentrate on the basic digital I/O port features.

As shown in Figure 1.2, each port has three registers associated with it:

- Data Register (PORTx) used to write output data to the port,
- Data Direction Register (DDRx) used to set a specific port pin to either output (1) or input (0), and
- Input Pin Address (PINx) used to read input data from the port.

Figure 1.2b describes the settings required to configure a specific port pin to either input or output. If selected for input, the pin may be selected for either an input pin or to operate in the high-impedance (Hi-Z) mode. In Hi-Z mode, the input appears as high impedance to a particular pin. If selected for output, the pin may be further configured for either logic low or logic high.


Port x Data Register—PORTx
7
0
Port x Data Direction Register—DDRx
7
0
Port x Input Pins Address—PINx
7
0

(a)

DDxn	PORTxn	I/O	Comment
0	0	Input	Tri-state (Hi-Z)
0	1	Input	Source current if externally pulled low
1	0	Output	Output Low (Sink)
1	1	Output	Output High (Source)

x: port designator (A, B, C, D)
n: pin designator (0–7)

(b)

Figure 1.2: ATmega164 port configuration registers: (a) port-associated registers and (b) port pin configuration.

Port pins are usually configured at the beginning of a program for either input or output, and their initial values are then set. Usually, all eight pins for a given port are configured simultaneously, even if all eight pins may not be used during a program execution. A code example to configure a port is shown below. Note that because we are using the C programming language with a compiler include file, the register contents are simply referred to by name. Note that the data direction register (DDRx) is first used to set the pins as either input or output, and then the data register (PORTx) is used to set the initial value of the output port pins. It is a good design practice to configure unused microcontroller pins as outputs. This prevents them from acting as a source for noise input.

```
//*************************************************************
//initialize_ports: provides initial configuration for I/O ports
//*************************************************************

void initialize_ports(void)
{
DDRA=0xfc;  //set PORTA[7:2] as output, PORTA[1:0]
            //as input (1111_1100)
PORTA=0x03; //initialize PORTA[7:2] low, PORTA[1:0]
            //current source
DDRB=0xa0;  //PORTB[7:4] as output, set PORTB[3:0] as input
PORTB=0x00; //disable PORTB pull-up resistors
DDRC=0xff;  //set PORTC as output
PORTC=0x00; //initialize low
DDRD=0xff;  //set PORTD as output
PORTD=0x00; //initialize low
}
```

To read the value from a port pin configured as input, the following code could be used. Note the variable used to read the value from the input pins is declared as an unsigned char because both the port and this variable type are 8 bits wide. A brief introduction to programming in C is provided in Chapter 2.

```
unsigned char   new_PORTB;            //new values of PORTB

:
:
:

new_PORTB = PINB;                     //read PORTB
```

1.4 PERIPHERAL FEATURES INTERNAL SUBSYSTEMS

In this section, a brief overview of the peripheral features of the ATmega164 is provided. It should be emphasized that these features are the internal subsystems contained within the confines of the microcontroller chip. These built-in features allow complex and sophisticated tasks to be accomplished by the microcontroller.

1.4.1 TIME BASE

The microcontroller is a complex synchronous state machine. It responds to program steps in a sequential manner as dictated by a user-written program. The microcontroller sequences through a predictable fetch, decode, and execute sequence. Each unique assembly language program instruction issues a series of signals to control the microcontroller hardware to accomplish instruction related operations.

The speed at which a microcontroller sequences through these actions is controlled by a precise time base, called the clock. The clock source is routed throughout the microcontroller to be used as a common time base for all peripheral subsystems. The ATmega164 may be clocked internally, using a user-selectable resistor capacitor (RC) time base, operating at approximately 8 MHz or externally using a timing crystal or resonator. The RC internal time base is selected using programmable fuse bits. We show its use in the application section.

To provide for a wider range of frequency selections, an external source may be used. The external time sources, in order of increasing accuracy and stability, are an external RC network, a ceramic resonator, or a crystal oscillator. The system designer chooses the time base frequency and clock source device appropriate for the application at hand. Generally speaking, a microcontroller is operated at the lowest possible frequency for a given application since clock speed is linearly related to power consumption. The clock source may be divided down by a user selectable value of 1, 2, 4, 8, 16, 32, 64, 128, or 256. The clock divide by value is set into the clock prescaler select (CLKPS) register or a divide by 8 value may be set with a fuse.

1.4.2 TIMING SUBSYSTEM

The ATmega164 is equipped with a complement of timers that allows the user to generate a precision output signal, measure the characteristics (period, duty cycle, frequency) of an incoming digital signal, or count external events. Specifically, the ATmega164 is equipped with two 8-bit timer/counters and one 16-bit counter. We discuss the operation, programming, and application of the timing system in Chapter 6.

1.4.3 PULSE WIDTH MODULATION CHANNELS

A pulse width modulated (PWM) signal is characterized by a fixed frequency and a varying duty cycle. Duty cycle is the percentage of time a repetitive signal is logic high during the signal period. It may be formally expressed as

$$\text{duty cycle } (\%) = (\text{on time/period}) \times (100\%).$$

The ATmega164 family is equipped with up to six PWM channels. The PWM channels coupled with the flexibility of dividing the time base down to different PWM subsystem clock source frequencies allow the user to generate a wide variety of PWM signals, from relatively high-frequency, low-duty cycle signals to relatively low-frequency, high-duty cycle signals. PWM signals are used in a wide variety of applications, including controlling the position of a servo motor and controlling the speed of a DC motor. We discuss the operation, programming, and application of the PWM system in Chapter 6.

1.4.4 SERIAL COMMUNICATIONS

The ATmega164 is equipped with a host of different serial communication subsystems, including the universal synchronous and asynchronous serial receiver and transmitter (USART), the serial peripheral interface (SPI), and the two-wire serial interface (TWI). What all of these systems have in common is the serial transmission of data. In a serial communications transmission scheme, data are sent a single bit at a time from transmitter to receiver.

Serial USART

The serial USART is used for full duplex (two-way) communication between a receiver and transmitter. This is accomplished by equipping the ATmega164 with independent hardware for the transmitter and receiver. The USART is typically used for asynchronous communication. That is, there is not a common clock between the transmitter and receiver to keep them synchronized with one another. To maintain synchronization between the transmitter and receiver, framing start and stop bits are used at the beginning and end of each data byte in a transmission sequence.

The ATmega164 USART is quite flexible. It has two independent channels designated USART0 and USART1. It has the capability to be set to a variety of data transmission rates known as the baud (bits per second) rate. The USART may also be set for data bit widths of 5–9 bits with one or two stop bits. Furthermore, the ATmega164 is equipped with a hardware-generated parity bit (even or odd) and parity check hardware at the receiver. A single parity bit allows for the detection of a single bit error within a byte of data. The USART may also be configured to operate in a synchronous mode. The flexible configuration of the USART system allows it to be used with a wide variety of peripherals including serial configured liquid crystal displays (LCDs) and a speech chip. The USART is also used to communicate microcontroller status to a host PC. We discuss the operation, programming, and application of the USART in Chapter 3.

Serial Peripheral Interface

The ATmega164 serial peripheral interface (SPI) can also be used for two-way serial communication between a transmitter and a receiver. In the SPI system, the transmitter and receiver share a common clock source. This requires an additional clock line between the transmitter and receiver but allows for higher data transmission rates, as compared with the USART.

The SPI may be viewed as a synchronous 16-bit shift register with an 8-bit half residing in the transmitter and the other 8-bit half residing in the receiver. The transmitter is designated the master because it provides the synchronizing clock source between the transmitter and the receiver. The receiver is designated as the slave. The SPI may be used to communicate with external peripheral devices such as large seven segment displays, digital-to-analog converts (DACs), and LED strips. We discuss the operation, programming, and application of the SPI in Chapter 3.

Two-Wire Serial Interface

The TWI subsystem allows the system designer to network a number of related devices (microcontrollers, transducers, displays, memory storage, etc.) together into a system using a two-wire interconnecting scheme. The TWI allows a maximum of 128 devices to be connected together. Each device has its own unique address and may both transmit and receive over the two-wire bus at frequencies up to 400 kHz. This allows the device to freely exchange information with other devices in the network within a small area.

1.4.5 ANALOG-TO-DIGITAL CONVERTER

The ATmega164 is equipped with an eight-channel ADC subsystem. The ADC converts an analog signal from the outside world into a binary representation suitable for use by the microcontroller. The ATmega164 ADC has 10-bit resolution. This means that an analog voltage between 0 and 5 V will be encoded into one of 1024 binary representations between $(000)_{16}$ and $(3FF)_{16}$. This provides the ATmega164 with a voltage resolution of approximately 4.88 mV. It is important to emphasize the ADC clock frequency must be set to a value between 50 and 200 kHz to obtain accurate results. We discuss the operation, programming, and application of the ADC in Chapter 4.

1.4.6 INTERRUPTS

The normal execution of a program follows a designated sequence of instructions. However, sometimes, this normal sequence of events must be interrupted to respond to high-priority faults, and status generated from both inside and outside the microcontroller. When these higher-priority events occur, the microcontroller must temporarily suspend its normal operation and execute event specific actions called an interrupt service routine. Once the higher priority event has been serviced, the microcontroller returns and continues processing the normal program.

The ATmega164 is equipped with a complement of 31 interrupt sources. Three of the interrupts are for external interrupt sources, whereas the remaining 28 interrupts support the efficient operation of peripheral subsystems aboard the microcontroller. We discuss the operation, programming, and application of the interrupt system in Chapter 5.

1.5 PHYSICAL AND OPERATING PARAMETERS

In this section, we present the physical layout and operating parameters of the ATmega164 microcontroller. As a system designer, it is important to know the various physical and operating parameter options available to select the best option for a given application.

1.5.1 PACKAGING

The ATmega164 comes in three different packaging styles: a 40-pin plastic dual in-line package (DIP), a 44-lead thin quad flat pack package, and a 44-pad quad flat nonlead/microlead frame package. The Pinout Diagram for the different packaging options are provided in Figure 1.3. The DIP package is especially useful for breadboard layout and first prototypes (e.g., capstone design projects).

1.5.2 POWER CONSUMPTION

The ATmega164 operates at supply voltages ranging from 1.8–5.5 VDC. In the application examples that follow, we use a standard laboratory 5-VDC power supply and also discuss a method of providing a 5-VDC supply using an off-the-shelf 9-VDC battery.

The current draw for the microcontroller is quite low. For example, when actively operating at 1 MHz from a 1.8 VDC power source, the current draw is 0.4 mA. In the power-down mode, the microcontroller draws less than 0.1 μA of current from the voltage source [Microchip].

To minimize power consumption, the microcontroller can be placed into various low-current sleep modes. There are six different sleep modes available to the system designer. The microcontroller is placed in sleep mode using the SLEEP command and "wakened" from SLEEP when an interrupt occurs. Additionally, power consumption can be further reduced by operating the microcontroller at the lowest practical clock frequency for a given application.

1.5.3 SPEED GRADES

The ATmega164 operates at several different speed grades. The maximum operating frequency with a supply voltage of 1.8 VDC is 4 MHz, 10 MHz at 2.7 VDC, and 20 MHz at 4.5 VDC. The maximum supply voltage at each of these operating frequencies is 5.5 VDC. The operating speed of the microcontroller is set by the time base chosen for the processor. One might believe that faster microcontroller operation is always better. This is not the case. The system designer must determine the minimum practical speed of microcontroller operation as the microcontroller's power consumption is directly related to operating speed. That is, the faster the oper-

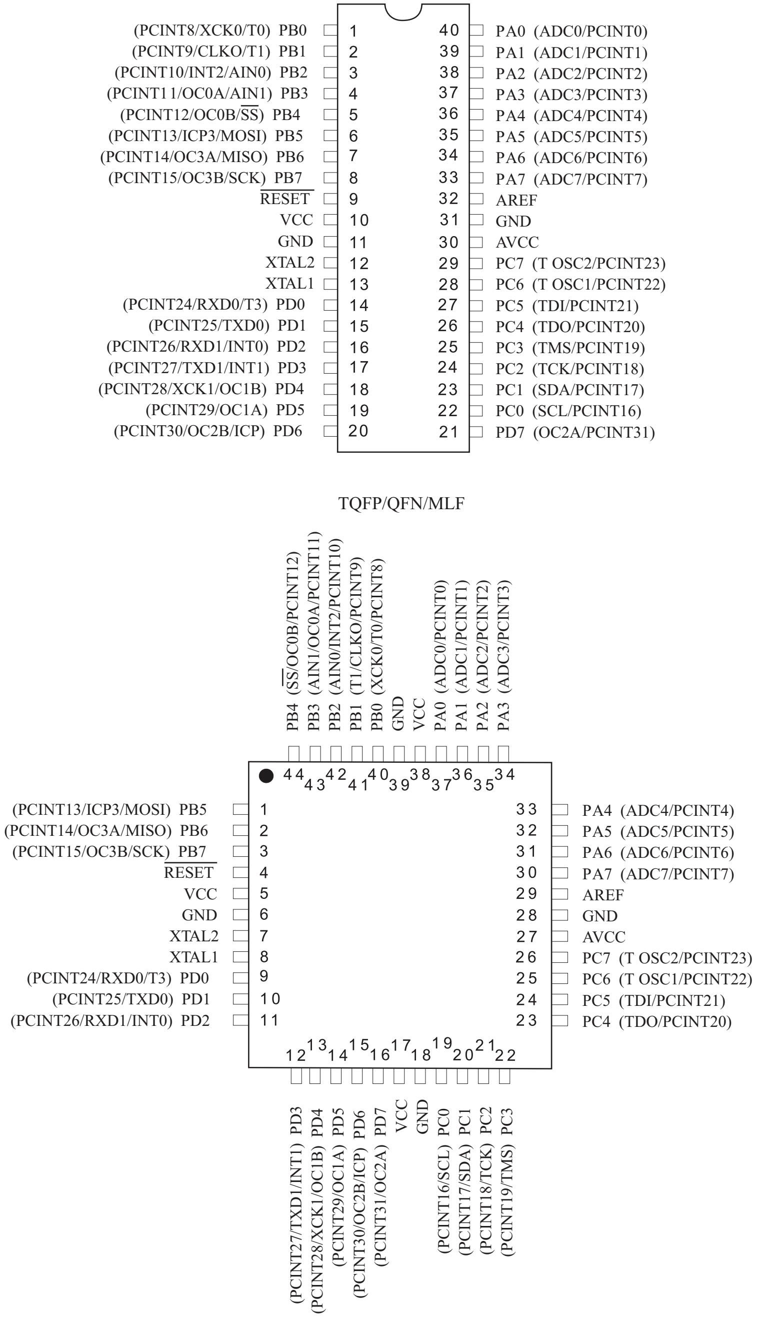


Figure 1.3: Microchip AVR ATmega164 Pinout Diagram: (a) 40-pin plastic DIP and (b) thin quad flat pack/microlead frame. Figure used with permission of Microchip.

ating speed of the microcontroller, the higher its power consumption. This becomes especially critical in portable, battery-operated embedded systems applications with a limited operation time [Microchip].

That completes our brief introduction to the features of the ATmega164. In the Application section, we apply what we have learned in developing a testbench for the ATmega164.

1.6 EXTENDED EXAMPLE: ATMEGA164 TESTBENCH

In this application, we present the hardware configuration of a barebones Testbench and a basic software framework to get the system up and operating. The purpose of the Testbench is to illustrate the operation of selected ATmega164 subsystems, working with various I/O devices. More importantly, the Testbench will serve as a template to develop your own applications.

We connect eight debounced tact switches to PORTB and an eight-channel tristate light-emitting diode (LED) array to PORTA. The software checks for a status change on PORTB. When the user depresses one of the tact switches, the ATmega164 will detect the status change and the corresponding LED on PORTA will transition from red to green.

1.6.1 HARDWARE CONFIGURATION

Provided in Figure 1.4 is the basic hardware configuration for the Testbench. We will discuss in detail the operation of the I/O devices in Chapter 6.

PORTA is configured with eight tact (momentary) switches with accompanying debouncing hardware. We discuss the debounce circuit in detail in Chapter 7. PORTB is equipped with an eight-channel tristate LED indicator. For a given port pin, the green LED will illuminate for a logic high, the red LED for a logic low, and no LEDs for a tristate high-impedance state. We discuss this circuit in detail in Chapter 7.

Aside from the input hardware on PORTB and the output display hardware on PORTA of the controller, there are power (pins 10, 30, and 32) and ground (pins 11 and 31) connections. A standard 5-VDC power supply may be used for the power connections. For portable applications, a 9-VDC battery equipped with a 5-VDC regulator (LM340-05 or uA7805) may be used as a power source. Pins 9–11 have a resistor (1 Mohm), two capacitors (1.0 μF), and a tact switch configured to provide a reset switch for the microcontroller. We use a ZTT 10-MHz ceramic resonator as the time base for the Testbench. It is connected to pins 12 (XTAL2) and 13 (XTAL1) of the ATmega164.

1.6.2 SOFTWARE CONFIGURATION

The Testbench software is provided below. The program contains the following sections:

- Comments
- Include Files: we have included the Atmel AVR Studio include file for the ATmega (avr/io.h). This file provides the software link between the names of the ATmega164 hard-

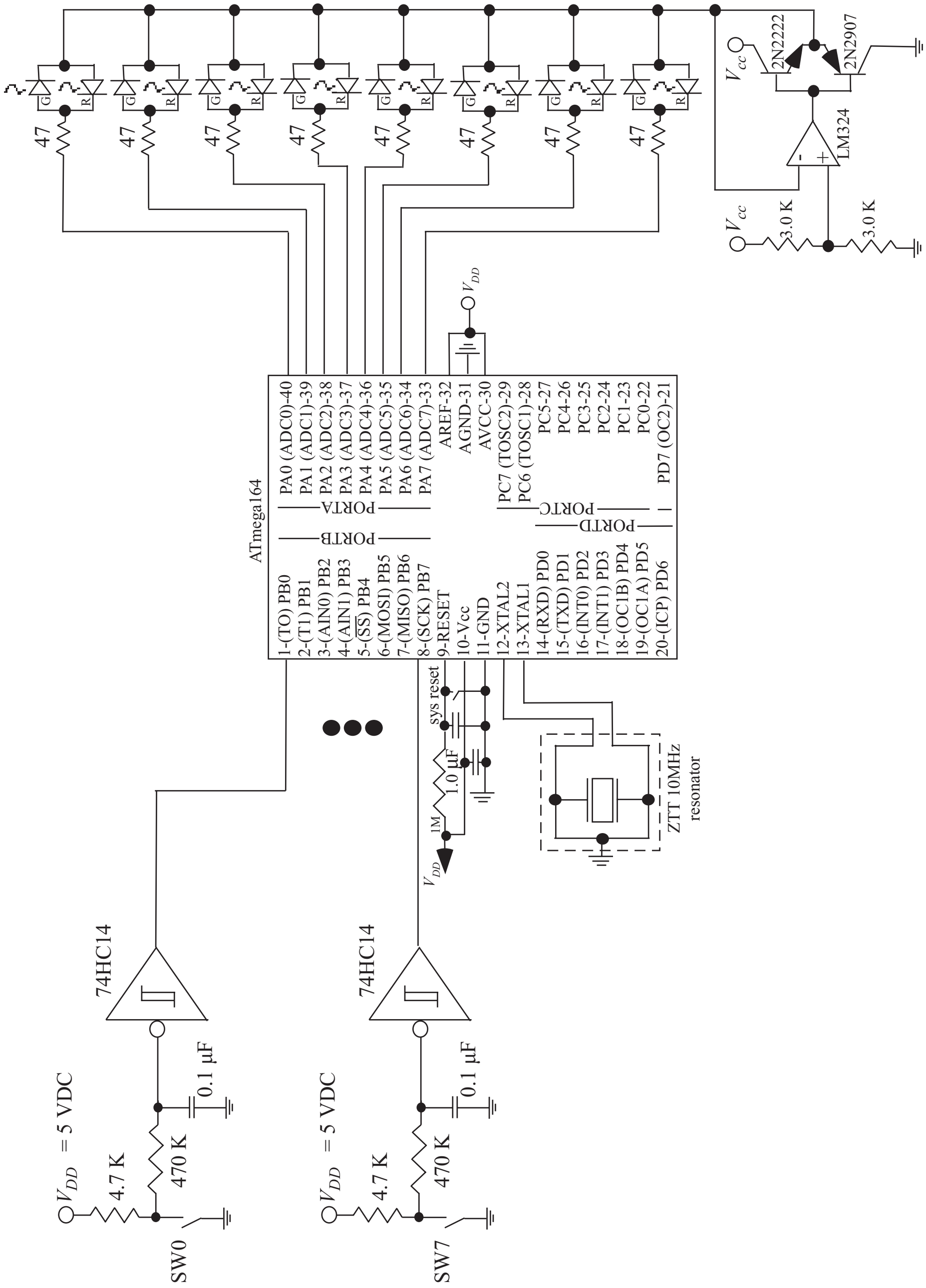


Figure 1.4: ATmega164 testbench hardware.

ware registers and the actual hardware locations. When a register is used by name in the program, reference is made to the contents of that register.

- Function Prototypes
- Global Variables
- Main Program: We begin the main program by calling the function to initialize the ports and then enter a continuous loop. Within the loop body, the ATmega164 monitors for a status change on PORTB. When the user depresses one of the tact switches connected to PORTB, a change of status is detected and the appropriate LED is illuminated on PORTA.
- Function Definition

We discuss details of program construction and programming in the next chapter.

```
//*************************************************************
//file name: testbench.c
//function: provides test bench for MICROCHIP AVR ATmega164 controller
//target controller: MICROCHIP ATmega164
//
//MICROCHIP AVR ATmega164 Controller Pin Assignments
//Chip Port Function I/O Source/Dest Asserted Notes
//Pin 1 PB0 to active high RC debounced switch
//Pin 2 PB1 to active high RC debounced switch
//Pin 3 PB2 to active high RC debounced switch
//Pin 4 PB3 to active high RC debounced switch
//Pin 5 PB4 to active high RC debounced switch
//Pin 6 PB5 to active high RC debounced switch
//Pin 7 PB6 to active high RC debounced switch
//Pin 8 PB7 to active high RC debounced switch
//Pin 9 Reset
//Pin 10 VDD
//Pin 11 Gnd
//Pin 12 Resonator
//Pin 13 Resonator
//Pin 14 PD0 to tristate LED indicator
//Pin 15 PD1 to tristate LED indicator
//Pin 16 PD2 to tristate LED indicator
//Pin 17 PD3 to tristate LED indicator
//Pin 18 PD4 to tristate LED indicator
```

```
//Pin 19 PD5 to tristate LED indicator
//Pin 20 PD6 to tristate LED indicator
//Pin 21 PD7 to tristate LED indicator
//Pin 22 PC0
//Pin 23 PC1
//Pin 24 PC2
//Pin 25 PC3
//Pin 26 PC4
//Pin 27 PC5
//Pin 28 PC6
//Pin 29 PC7
//Pin 30 AVcc to VDD
//Pin 31 AGnd to Ground
//Pin 32 ARef to Vcc
//Pin 33 PA7
//Pin 34 PA6
//Pin 35 PA5
//Pin 36 PA4
//Pin 37 PA3
//Pin 38 PA2
//Pin 39 PA1
//Pin 40 PA0
//author: Steven Barrett and Daniel Pack
//created: July 12, 2007
//last revised: August 13, 2016
//***************************************************************

//Include Files: choose the appropriate include file depending on
//the compiler in use - comment out the include file not in use.

//include file(s) for JumpStart C for AVR Compiler****************
#include<iom164pv.h>                    //contains reg definitions

//include file(s) for the Atmel Studio gcc compiler
//#include <avr/io.h>                   //contains reg definitions


//function prototypes********************************************
```

```
void initialize_ports(void);           //initializes ports

//main program***************************************************
//global variables
unsigned char   old_PORTB = 0x00;   //present value of PORTB
unsigned char   new_PORTB;          //new values of PORTB void main(void)
{
initialize_ports();                 //initialize ports

while(1)
  {
  //main loop   new_PORTB = PINB;   //read PORTB

  if(new_PORTB != old_PORTB){       //process change
                                    //in PORTB input
    switch(new_PORTB){              //PORTB asserted high

           case 0x01:               //PB0 (0000_0001)
             PORTD=0x00;            //turn off all LEDs PORTD
             PORTD=0x01;            //turn on PD0 LED (0000_0001)
             break;

           case 0x02:               //PB1 (0000_0010)
             PORTD=0x00;            //turn off all LEDs PORTD
             PORTD=0x02;            //turn on PD1 LED (0000_0010)
             break;

           case 0x04:               //PB2 (0000_0100)
             PORTD=0x00;            //turn off all LEDs PORTD
             PORTD=0x04;            //turn on PD2 LED (0000_0100)
             break;

           case 0x08:               //PB3 (0000_1000)
             PORTD=0x00;            //turn off all LEDs PORTD
             PORTD=0x08;            //turn on PD3 LED (0000_1000)
             break;

           case 0x10:               //PB4 (0001_0000)
             PORTD=0x00;            //turn off all LEDs PORTD
             PORTD=0x10;            //turn on PD4 LED (0001_0000)
             break;

           case 0x20:               //PB5 (0010_0000)
             PORTD=0x00;            //turn off all LEDs PORTD
             PORTD=0x20;            //turn on PD5 LED (0010_0000)
             break;

           case 0x40:               //PB6 (0100_0000)
             PORTD=0x00;            //turn off all LEDs PORTD
             PORTD=0x40;            //turn on PD6 LED (0100_0000)
             break;

           case 0x80:               //PB7 (1000_0000)
             PORTD=0x00;            //turn off all LEDs PORTD
             PORTD=0x80;            //turn on PD7 LED (1000_0000)
             break;
   default:;                  //all other cases
           }                        //end switch(new_PORTB)
         }                          //end if new_PORTB

      old_PORTB=new_PORTB;          //update PORTB
  }                                 //end while(1)
}                                   //end main
//***************************************************************
//function definitions
//***************************************************************

//***************************************************************
//initialize_ports: provides initial configuration for I/O ports
//***************************************************************

void initialize_ports(void)
{
DDRA=0xff;                          //set PORTA[7:0] as output
PORTA=0x00;                         //initialize PORTA[7:0] low

DDRB=0x00;                          //PORTB[7:0] as input
PORTB=0x00;                         //disable PORTB
```

```
                                    //pull-up resistors

DDRC=0xff;                          //set PORTC as output
PORTC=0x00;                         //initialize low

DDRD=0xff;                          //set PORTD as output
PORTD=0x00;                         //initialize low
}
//*****************************************************************
```

1.7 PROGRAMMING THE ATMEGA164

Programming the ATmega164 requires several hardware and software tools. For software tools a compiler and device programming support is required. Throughout the book we provide examples using both the ImageCraft JumpStart C for AVR compiler (www.imagecraft.com) and also the Atmel Studio gcc compiler (www.microchip.com). We use the Atmel Studio software suite with the Microchip AVR Dragon programmer to download and program the microcontroller. The connection between the host computer and the Microchip AVR Dragon is shown in Figure 1.5 with detailed instructions provided below. We use eight LEDs connected to PORTD as shown in Figure 1.6. We discuss the testbench circuit in some detail in Chapter 7. Also note the use of an external 20 MHz ceramic resonator as the microcontroller time base.

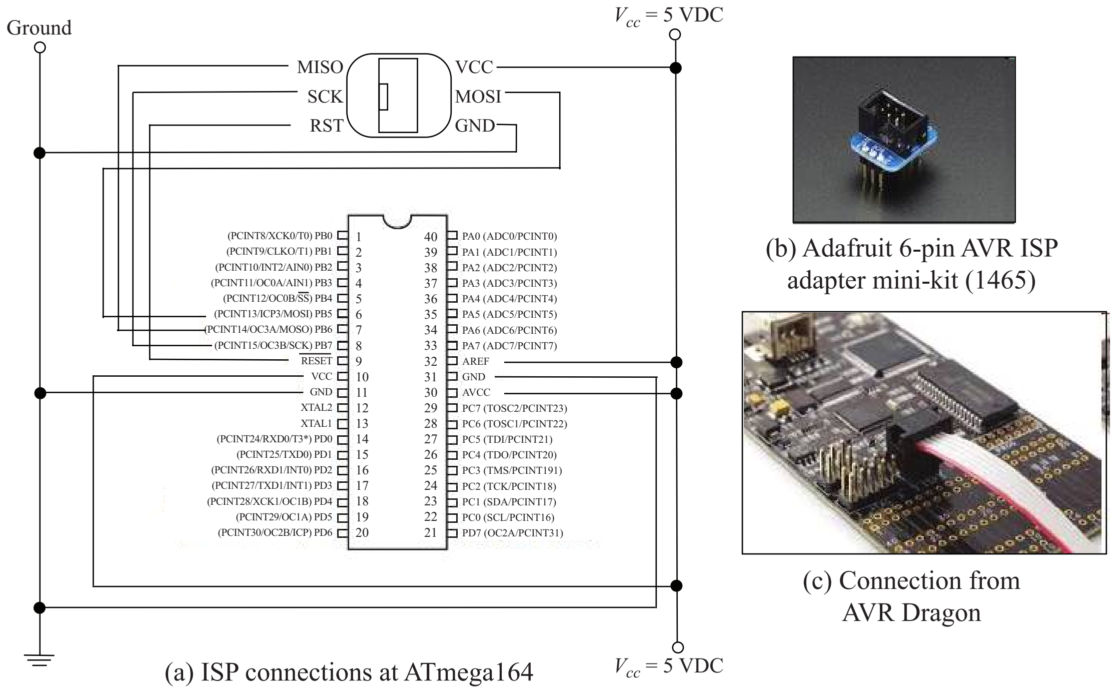
1.7.1 IMAGECRAFT JUMPSTART C FOR AVR COMPILER DOWNLOAD, INSTALLATION, AND ATMEGA164 PROGRAMMING

Throughout the text, we provide examples using the ImageCraft JumpStart C for AVR compiler. This is an excellent, well-supported, user-friendly compiler. The compiler is available for purchase and download at www.imagecraft.com. Details on compiler download and configuration are provided there. ImageCraft allows a 45-day compiler "test drive" before securing a software license. One of the authors (sfb) has used variants of this compiler for over a decade on multiple Microchip AVR products. You can expect prompt, courteous service from the company. There are other excellent compilers available. The compiler is used to translate the source file (testbench.c) into machine language (testbench.hex) for loading into the ATmega164. We use Atmel's AVR Studio to load the machine code into the ATmega164.

1. Download and install ImageCraft JumpStart C for AVR compiler.

2. Create a new project and select ImageCraft empty project. Press Next.

3. Provide a descriptive project title and browse to the desired folder location.



(a) ISP connections at ATmega164

(b) Adafruit 6-pin AVR ISP adapter mini-kit (1465)

(c) Connection from AVR Dragon

Figure 1.5: Programming the ATmega164 with the AVR dragon (www.adafruit.com, www.microchip.com).

4. Select target as ATmega164. Details of the microcontroller will populate drop-down windows. Press OK.

1.7.2 ATMEL STUDIO DOWNLOAD, INSTALLATION, AND ATMEGA164 PROGRAMMING

1. Download and install Atmel Studio version 7.0 or later.
2. Connect the AVR Dragon to the host PC via a USB cable. The AVR Dragon drivers should automatically install.
3. Configure hardware:
   - Configure the Adafruit 6-pin AVR ISP adaptor mini-kit (1465) as shown in Figure 1.5.
   - Connect the Adafruit 6-pin AVR ISP adaptor to the AVR Dragon via 6-pin socket IDC cable.
   - Connect the Adafruit 6-pin AVR ISP adaptor to ATmega164P target as shown.

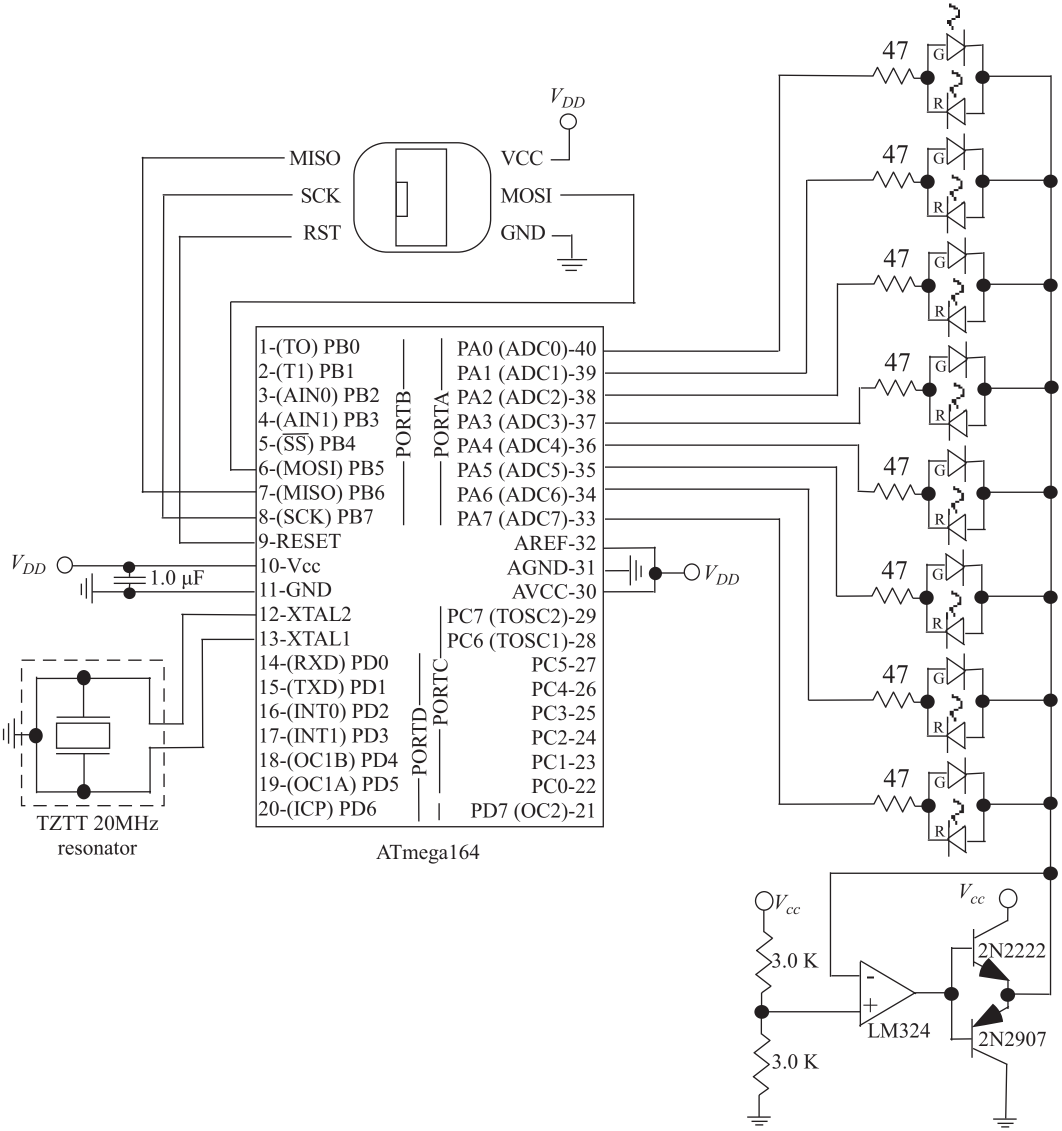


Figure 1.6: LED connections to ATmega164.

4. Program Compiling and Loading

    - Start Atmel Studio.
    - If using the gcc compiler: File – > New – > Project – > GCC Executable Project – > <filename>
    - If using the gcc compiler: Write program
    - If using the gcc compiler: Build program
    - Tools Device Programming
        - Dragon
        - ATmega164P
        - Interface: ISP
        - Insure target chip has 5 VDC applied

    Apply

        - Read signature, read target voltage

    - Memories: Flash: Program
    - Fuses: Ext. Crystal Osc. 8.0 MHz: Program

1.8 SOFTWARE PORTABILITY

The software techniques discussed in the textbook are for the ATmega164; however, the developed software may be easily ported for use with other Microchip AVR microcontrollers. To ease the transition to another microcontroller, it is suggested using a direct bit assignment technique. Instead of setting an entire register content at once, selected bits may be set. The individual bit definitions for the ATmega164 are provided in the Appendices.

For example, to set the UCSRB register, the following individual bit assignments may be used:

```
UCSRB = (1<<TXEN)|(1<<RXEN);       //Enable transmit and receive

as opposed to:

UCSRB = 0x08;                      //Enable transmit and receive
```

When transporting code, the header file for the specific microcontroller must be used, and also, the interrupt vector numbers may require change.

1.9 APPLICATION

In this example we program a rain gauge indicator using the testbench hardware LEDs on PORTD. LEDs are sequentially illuminated with a 1-s delay between PORTD changes. We use a simple, yet inaccurate, method to generate the delay. We provide a more sophisticated and accurate method using interrupts in Chapter 5.

```
//*************************************************************
//file name: chp1_app
//function: provides test bench for MICROCHIP AVR ATmega164 controller
//target controller: MICROCHIP ATmega164
//
//Microchip AVR ATmega164 Controller Pin Assignments
//Chip Port Function I/O Source/Dest Asserted Notes
//Pin 1 - 7: PB0 to PB7 set to output: unused
//Pin 9 Reset
//Pin 10 VDD
//Pin 11 Gnd
//Pin 12 and 13 Resonator
//Pin 14 - 21: PD0 to PD7 output to tristate LED indicator
//Pin 22 - 29: PC0 to PC7 set to output: unused
//Pin 30 AVcc to VDD
//Pin 31 AGnd to Ground
//Pin 32 ARef to Vcc
//Pin 33 - 40: PA7 to PA0 set to output: unused
//
//External resonator: 20 MHz
//
//author: Steven Barrett and Daniel Pack
//created: Aug 16, 2016
//last revised: Aug 16, 2016
//***************************************************************

//function prototypes********************************************
void initialize_ports(void);       //initializes ports
void delay_100ms(void);
void delay_1s(void);

//Include Files: choose the appropriate include file depending on
//the compiler in use - comment out the include file not in use.
```

```
//include file(s) for JumpStart C for AVR Compiler****************
#include<iom164pv.h>                   //contains reg definitions

//include file(s) for the Microchip Studio gcc compiler
//#include <avr/io.h>                  //contains reg definitions


int main(void)
{
unsigned char count = 0;

initialize_ports();

for(count = 0; count <=2; count++)
  {
  PORTD = 0x00;
  delay_1s();
  PORTD = 0x01;
  delay_1s();
  PORTD = 0x03;
  delay_1s();
  PORTD = 0x07;
  delay_1s();
  PORTD = 0x0F;
  delay_1s();
  PORTD = 0x1F;
  delay_1s();
  PORTD = 0x3F;
  delay_1s();
  PORTD = 0x1F;
  delay_1s();
  PORTD = 0x7F;
  delay_1s();
  PORTD = 0xFF;
  delay_1s();
  }
}
```

```
//***************************************************************
//function definitions
//***************************************************************

//***************************************************************
//initialize_ports: provides initial configuration for I/O ports
//***************************************************************
void initialize_ports(void)
{
DDRA=0xff;                         //set PORTA[7:0] as output
PORTA=0x00;                        //initialize low

DDRB=0xff;                         //PORTB[7:0] as output
PORTB=0x00;                        //initialize low

DDRC=0xff;                         //set PORTC as output
PORTC=0x00;                        //initialize low

DDRD=0xff;                         //set PORTD as output
PORTD=0x00;                        //initialize low

}

//***************************************************************
//delay_100ms: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 20 MHz
//  - 100 ms delay requires 2M clock cycles
//  - nop requires 1 clock cycle to execute
//***************************************************************
void delay_100ms(void)
{
unsigned int i,j;

for(i=0; i < 2000; i++)
  {
  for(j=0; j < 1000; j++)
    {
    asm("nop");                           //inline assembly
    }                                      //nop: no operation
  }                                        //requires 1 clock cycle
}

//*****************************************************************
//delay_1s: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 20 MHz
//  - 100 ms delay requires 2M clock cycles
//  - nop requires 1 clock cycle to execute
//  - call 10 times for 1s delay
//*****************************************************************

void delay_1s(void)
{
unsigned int i;

for(i=0; i< 10; i++)
  {
  delay_100ms();
  }
}

//*****************************************************************
```

1.10 LABORATORY EXERCISE: TESTBENCH

Write a program in C that consists of four functions:

- The first function will display an incrementing binary count from 0–255 on PORTD.
- The second function will display a decrementing count on the PORTD.
- The third function will provide a rain gauge display on the PORTD. That is, LED[0] will illuminate, then LED [1:0], LED [2:0], etc.
- The fourth function will be called delay_30ms. It should provide a delay of at least 30 ms.
- Call each function sequentially from the main program. The delay function should be called every time the PORTD is updated with a new value.

1.11 SUMMARY

In this chapter, we provided a brief overview of the ATmega164 microcontroller, a representative sample of the AVR microcontrollers. Information presented in this chapter can be readily applied to other microcontrollers in the AVR microcontroller line. We then provided the Testbench hardware and software that we use throughout the text to illustrate peripheral subsystem operation aboard the ATmega164. In upcoming chapters, we provide additional details on selected ATmega164 subsystems.

1.12 REFERENCES AND FURTHER READING

Atmel 8-bit AVR Microcontroller with 16/32/64/128 K Bytes In-System Programmable Flash, ATmega164A, ATmega164PA, ATmega324A, ATmega324PA, ATmega644A, ATmega644PA, ATmega1284, ATmega1284P, 8272C-AVR-06/11, data sheet: 8272G-AVR-01/15, Atmel, San Jose, CA, 2015. 2, 11, 13

Barrett, S. F. and Pack, D. J. *Microcontroller Fundamentals for Engineers and Scientists*, Morgan & Claypool, San Rafael, CA, 2006. DOI: 10.2200/s00025ed1v01y200605dcs001. 1

Hennessy, J. and Patterson, D. *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufman, San Francisco, CA, 2003. 2

1.13 CHAPTER PROBLEMS

1. What is a RISC processor?
2. How does the ATmega164 interact with the external world?
3. What are the different methods of applying a clock source to the ATmega164? List the inherent advantages of each type.
4. Describe the three different types of memory components aboard the ATmega164. What is each used for?
5. Describe the three registers associated with each port.
6. With a specific port, can some port pins be declared as output pins while others as input pins? Explain.
7. What are the trade-offs of programming a microcontroller using assembly language vs. C?
8. Describe the serial communication features aboard the ATmega164. Provide a suggested application for each.
9. What is the purpose of the ADC system?
10. What is ADC resolution? What is the resolution of the ADC system onboard the ATmega164?

11. What is the purpose of the interrupt system?

12. What is the purpose of the PWM system?

13. What is the best clock speed to operate the ATmega164 at for a specific application?

14. Sketch a flow chart for the testbench.c program.

15. In the application example, we counted no operations (nop) as a method to generate a delay. Why is this technique inherently inaccurate? How can it be improved?

CHAPTER 2

Programming

Objectives: After reading this chapter, the reader should be able to:

- describe the key components of a program;
- specify the size of different variables within the C programming language;
- define the purpose of the main function;
- explain the importance of using functions within a program;
- write functions that pass parameters and return variables;
- describe the purpose of a header file;
- discuss different programming constructs used for program control and decision processing; and
- write programs for use with the ATmega164.

2.1 OVERVIEW

To a novice user, programming a microcontroller may appear mysterious, complicated, overwhelming, or difficult. When faced with an unfamiliar task, one often does not know where to start. The goal of this chapter is to provide a tutorial on how to begin microcontroller programming. We use a top-down design approach. We begin with the "big picture" of a typical program followed by an overview of the major pieces of the program. We then discuss the basics of the C programming language. Only the most fundamental concepts are covered. Throughout the chapter, we use examples to illustrate concepts and provide pointers to a number of excellent references for further reading.

2.2 THE BIG PICTURE

Most microcontrollers are programmed with some variant of the C programming language. The C programming language provides a nice balance between the programmer's control of the microcontroller hardware and time efficiency in program writing. The compiler software is hosted on a computer separate from the microcontroller. The job of the compiler is to transform the program provided by a programmer (filename.c and filename.h) into machine code (filename.hex or filename.elf) suitable for loading into the microcontroller.

Once the source files (filename.c and filename.h) are provided to the compiler, the compiler executes two steps to render the machine code. The first step is the compilation process. Here the program source files are transformed into assembly code (filename.asm). If the program source files contain syntax errors, the compiler reports these to the user. Syntax errors are reported for incorrect use of the C programming language. An assembly language program is not generated until the syntax errors have been corrected.

The assembly language source file (filename.asm) is then passed to the assembler, which transforms the assembly language source file (filename.asm) to machine code (filename.hex or folename.elf) suitable for loading to the Microchip processor.

In the next section, we discuss the components of a C program.

2.3 ANATOMY OF A PROGRAM

Programs written for a microcontroller have a fairly repeatable format. Slight variations exist but many follow the format shown below.

```
//Comments containing program information
// - file name:
// - author:
// - revision history:
// - compiler setting information:
// - hardware connection description to microcontroller pins
// - program description

//include files
#include<file_name.h>

//function prototypes
A list of functions and their format used within the program

//program constants
#define    TRUE   1
#define    FALSE  0
#define    ON     1
#define    OFF    0

//interrupt handler definitions
```

```
Used to link the software to hardware interrupt features

//global variables
Listing of variables used throughout the program

//main program

void main(void)
{

body of the main program

}

//function definitions
A detailed function body and definition
for each function used within the program
```

Let's take a closer look at each piece.

2.3.1 COMMENTS

Comments are typically used throughout programs to document what and how things were accomplished. The comments also help a programmer to reconstruct his or work at a later time. Imagine that you wrote a program a year ago for a project. You now want to modify that program for a new project. The comments will help you remember the key details of the program.

Comments are not compiled into machine code for loading into the microcontroller. Therefore, the comments will not fill up the memory of your microcontroller. Comments are indicated using double slashes (//). Anything from the double slashes to the end of a line is then considered a comment by a compiler. A multi-line comment can be constructed using a /∗ at the beginning of the comment and a ∗/ at the end of the comment. These are handy to block out portions of code during troubleshooting or providing multi-line comments.

At the beginning of the program, comments may be extensive. Comments may include some of the following information:

- file name,
- program author,
- revision history or a listing of the key changes made to the program,
- compiler setting information,

- hardware connection description to microcontroller pins, and
- detailed program description.

2.3.2 INCLUDE FILES

Often you need to add extra files to your project besides the main program. For example, most compilers require a "personality file" on the specific microcontroller that you are using. This file contains the name of each register used within the microcontroller. It also provides the link between a specific register's name used in the software and the actual register location within hardware. These files are typically called header files and their name ends with a ".h". Within the C compiler there may be other header files to include in your program such as the "math.h" file when programming with advanced math functions.

To include header files within a program, the following syntax is used:

```
//include files
#include<file_name1.h>
#include<file_name2.h>
```

2.3.3 FUNCTIONS

In the next chapter, we discuss in detail the top-down design, bottom-up implementation approach to design microcontroller based systems. In this approach, a microcontroller based project, including both hardware and software, is partitioned into systems, subsystems, etc. The idea is to take a complex project and break it into "doable" pieces with a defined action.

We use the same approach when writing computer programs. At the highest level, we have the main program which calls functions that perform predefined actions. When a function is called, program control is released from the main program to the function. Once the function is complete, program control reverts back to the main program.

Functions may in turn call other functions as shown in Figure 2.1. This approach results in a collection of functions that may be reused over and over again in various projects. Most importantly, the program is now subdivided into doable pieces, each with a defined action. This makes writing the program easier but also makes it simple to modify the program, since every action is at a known location.

There are three different pieces of code required to properly configure and call the function:

- function prototype,
- function call, and
- function body.

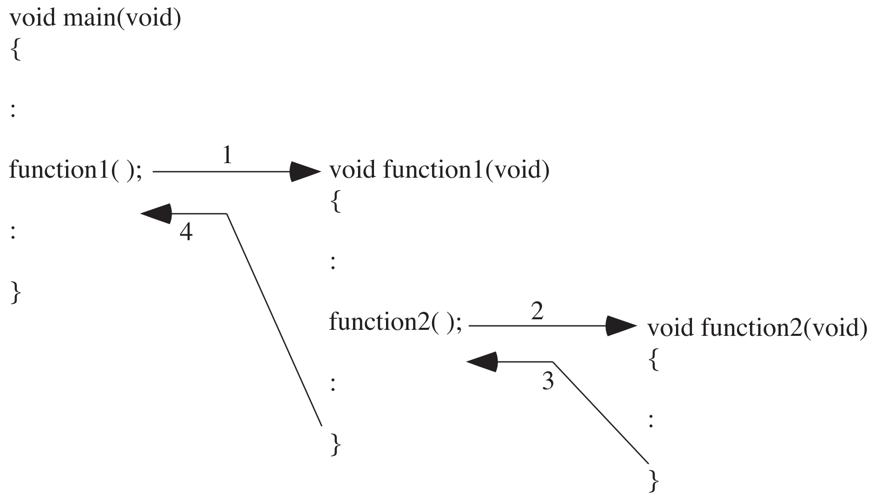


Figure 2.1: Function calling. The numbers provided in the diagram indicate the movement of program control.

Function prototypes are provided early in the program as previously shown in the program template. The function prototype shows the name of the function, variables required by the function, and the type of variable returned by the function.

The function prototype follows this format:

```
return_variable  function_name(required_variable1, required_variable2);
```

If the function does not require variables or sends back a value the word "void" is placed in the variable's position.

The **function call** is the code statement used within a program to execute the function. The function call consists of the function name and the actual arguments required by the function. If the function does not require arguments to be delivered to it for processing, the parenthesis containing the variable list is left empty.

The function call follows the following format:

```
function_name(required_variable1, required_variable2);
```

A function that requires no variables follows this format:

```
function_name( );
```

When the function call is executed by the program, program control is transferred to the function, the function is executed, and program control is then returned to the portion of the program that called it.

The **function body** is a self-contained "mini-program." The first line of the function body contains the same information as the function prototype: the name of the function, any variables required by the function, and any value returned by the function. The last line of the function contains a "return" statement. Here a value may be sent back to the portion of the program that called the function. The processing action of the function is contained within the open ({) and close brackets (}). If the function requires any variables within the confines of the function, they are declared next. These variables are referred to as local variables. A local variable is known only within the confines of a specific function. The actions required by the function follow the declaration of local variables.

The function prototype follows this format:

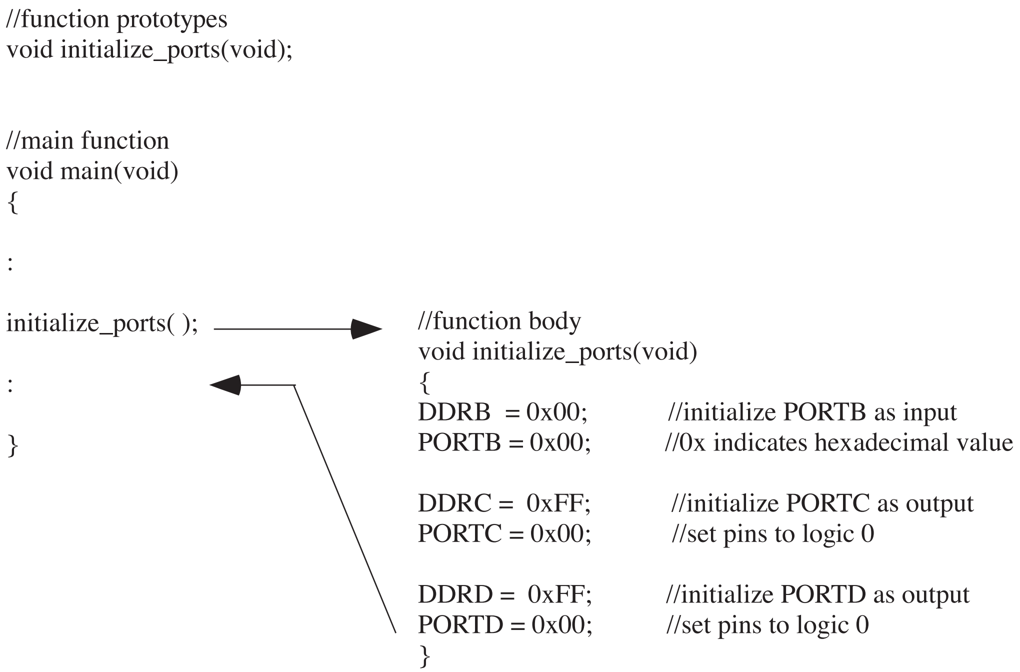
```
return_variable  function_name(required_variable1, required_variable2)
{
//local variables required by the function
unsigned int  variable1;
unsigned char variable2;

//program statements required by the function



//return variable
return return_variable;
}
```

Example: In this example, we describe how to configure the ports of the microcontroller to act as input or output ports. Briefly, associated with each port is a register called the data direction register (DDR). Each bit in the DDR corresponds to a bit in the associated PORT. For example, PORTB has an associated data direction register DDRB. If DDRB[7] is set to a logic 1, the corresponding port pin PORTB[7] is configured as an output pin. Similarly, if DDRB[7] is set to logic 0, the corresponding port pin is configured as an input pin.

During some of the early steps of a program, a function is called to initialize the ports as input, output, or some combination of both. This is illustrated in Figure 2.2.

2.3.4 PROGRAM CONSTANTS

The #define statement is used to associate a constant name with a numerical value in a program. It can be used to define common constants such as pi. It may also be used to give terms used within a program a numerical value. This makes the code easier to read. For example, the following constants may be defined within a program:



```
//function prototypes
void initialize_ports(void);


//main function
void main(void)
{

:

initialize_ports( );

:

}
```

```
//function body
void initialize_ports(void)
{
DDRB  = 0x00;           //initialize PORTB as input
PORTB = 0x00;           //0x indicates hexadecimal value

DDRC =  0xFF;           //initialize PORTC as output
PORTC = 0x00;           //set pins to logic 0

DDRD =  0xFF;           //initialize PORTD as output
PORTD = 0x00;           //set pins to logic 0
}
```

Figure 2.2: Configuring ports.

```
//program constants
#define    TRUE   1
#define    FALSE  0
#define    ON     1
#define    OFF    0
```

2.3.5 INTERRUPT HANDLER DEFINITIONS

Interrupts are functions that are written by the programmer but usually called by a specific hardware or software event during system operation. We discuss interrupts and how to properly configure them in an upcoming chapter.

2.3.6 VARIABLES

There are two types of variables used within a program: global variables and local variables. A global variable is available and accessible to all portions of the program, whereas a local variable is only known and accessible within the function where it is declared.

When declaring a variable in C, the number of bits used to store the operator is also specified. In Figure 2.3, we provide a list of common C variable sizes used with the ImageCraft

Type	Size (bytes)	Range
unsigned char	1	0..255
signed char	1	-128..127
unsigned int	2	0..65535
signed int	2	-32768..32767
float	4	+/-1.175e-38.. +/-3.40e+38
double	4	+/-1.175e-38.. +/-3.40e+38

Figure 2.3: C variable sizes used with the ImageCraft ICC AVR compiler (www.imagecraft.com).

JumpStart C for AVR compiler. The size of other variables such as pointers, shorts, longs, etc. are contained in the compiler documentation (www.imagecraft.com).

When programming microcontrollers, it is important to know the number of bits used to store a variable and where that variable will be assigned. For example, assigning the contents of an unsigned char variable, which is stored in 8 bits, to an 8-bit output port will have a predictable result. However, assigning an unsigned int variable, which is stored in 16 bits, to an 8-bit output port does not provide predictable results. It is wise to ensure your assignment statements are balanced for accurate and predictable results. The modifier "unsigned" indicates all bits will be used to specify the magnitude of the argument. Signed variables will use the left-most bit to indicate the polarity (±) of the argument.

A global variable is declared using the format provided below. The type of the variable is specified, followed by its name, and an initial value, if desired.

```
//global variables
unsigned int  loop_iterations = 6;
```

2.3.7 MAIN PROGRAM

The main program is the hub of activity for the entire program. The main program typically consists of program steps and function calls to initialize the processor followed by program steps to collect data from the environment external to the microcontroller, process the data and make decisions, and provide external control signals back to the environment based on the data collected.

2.4 FUNDAMENTAL PROGRAMMING CONCEPTS

In the previous section, we covered many fundamental concepts. In this section we discuss operators, programming constructs, and decision processing constructs to complete our fundamental overview of programming concepts.

2.4.1 OPERATORS

There are a wide variety of operators available in the C language. An abbreviated list of common operators are shown in Figures 2.4 and 2.5. The operators have been grouped by general category. The symbol, precedence, and brief description of each operator are presented. The precedence column indicates the priority of the operator in a program statement containing multiple operators. Only the fundamental operators are shown. For more information on this topic, see Barrett and Pack [2005].

General Operations

Within the general operations category are brackets, parenthesis, and the assignment operator. We have seen in an earlier example how bracket pairs are used to indicate the beginning and end of the main program or a function. They are also used to group statements in programming constructs and decision processing constructs. This is discussed in the next several sections.

The parenthesis is used to boost the priority of an operator. For example, in the mathematical expression $7 \times 3 + 10$, the multiplication operation is performed before the addition since it has a higher precedence. Parenthesis may be used to boost the precedence of the addition operation. If we contain the addition operation within parenthesis $7 \times (3 + 10)$, the addition will be performed before the multiplication operation and the overall operations yield a different result from the earlier expression.

The assignment operator ($=$) is used to assign the argument(s) on the right-hand side of an equation to the left-hand side variable. It is important to ensure that the left and the right-hand side of the equation have the same type of arguments. If not, unpredictable results may occur.

Arithmetic Operations

The arithmetic operations provide for basic math operations using the various variables described in the previous section.

Example: In this example, a function returns the sum of two unsigned int variables passed to the function.

General		
Symbol	Precedence	Description
{ }	1	Brackets, used to group program statements
( )	1	Parenthesis, used to establish precedence
=	12	Assignment

Arithmetic Operations		
Symbol	Precedence	Description
*	3	Multiplication
/	3	Division
+	4	Addition
-	4	Subtraction

Logical Operations		
Symbol	Precedence	Description
<	6	Less than
<=	6	Less than or equal to
>	6	Greater
>=	6	Greater than or equal to
==	7	Equal to
!=	7	Not equal to
&&	9	Logical AND
||	10	Logical OR

Figure 2.4: C operators. (Adapted from Barrett and Pack [2005].)

Bit Manipulation Operations		
Symbol	Precedence	Description
<<	5	Shift left
>>	5	Shift right
&	8	Bitwise AND
^	8	Bitwise exclusive OR
|	8	Bitwise OR

Unary Operations		
Symbol	Precedence	Description
!	2	Unary negative
~	2	One's complement (bit-by-bit inversion)
++	2	Increment
--	2	Decrement
type(argument)	2	Casting operator (data type conversion)

Figure 2.5: C operators (continued). (Adapted from Barrett and Pack [2005].)

```
unsigned int  sum_two(unsigned int variable1, unsigned int variable2)
{
unsigned int  sum;

sum = variable1 + variable2;

return sum;
}
```

Logical Operations

The logical operators provide Boolean logic operations. They can be viewed as comparison operators. One argument is compared against another using the logical operator provided. The result is returned as a logic value of one (1, true, high) or zero (0 false, low). The logical operators are used extensively in program constructs and decision processing operations to be discussed in the next several sections.

Bit Manipulation Operations

There are two general types of operations in the bit manipulation category: shifting operations and bitwise operations. Let's examine several examples.

Example: Given the following code segment, what will the value of variable2 be after execution?

```
unsigned char   variable1 = 0x73;
unsigned char   variable2;

variable2 = variable1 << 2;
```

Answer: Variable "variable1" is declared as an 8-bit unsigned char and assigned the hexadecimal value of $(73)_{16}$. In binary this is $(0111\_0011)_2$. The $<<$ 2 operator performs a left shift of the argument by two places. After two left shifts of $(73)_{16}$, the result is $(cc)_{16}$ and is assigned to the variable "variable2."

Note that the left and right shift operation is equivalent to multiplying and dividing the variable by a power of two.

The bitwise operators perform the desired operation on a bit-by-bit basis. That is, the least significant bit of the first argument is bit-wise operated with the least significant bit of the second argument and so on.

Example: Given the following code segment, what will the value of variable3 be after execution?

```
unsigned char   variable1 = 0x73;
unsigned char   variable2 = 0xfa;
unsigned char   variable3;

variable3 = variable1 & variable2;
```

Answer: Variable "variable1" is declared as an 8-bit unsigned char and assigned the hexadecimal value of $(73)_{16}$. In binary, this is $(0111\_0011)_2$. Variable "variable2" is declared as an 8-bit unsigned char and assigned the hexadecimal value of $(fa)_{16}$. In binary, this is $(1111\_1010)_2$. The bitwise AND operator is specified. After execution, variable "variable3," declared as an 8-bit unsigned char, contains the hexadecimal value of $(72)_{16}$.

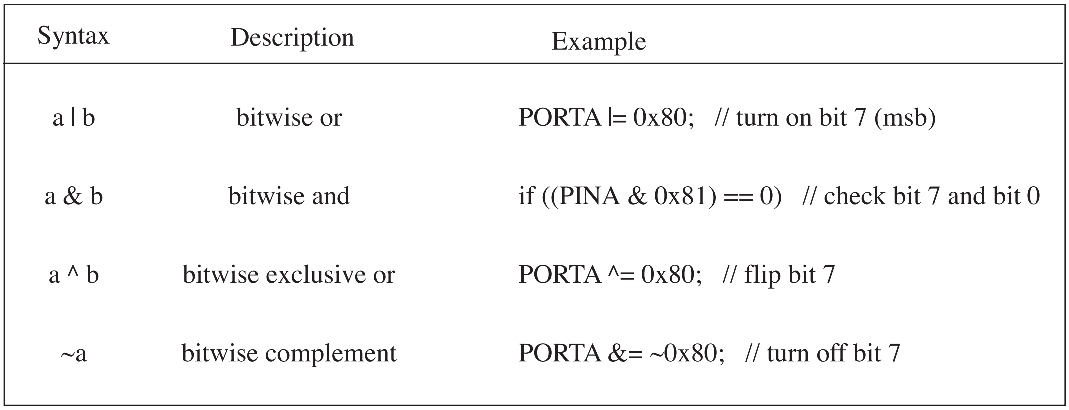
Unary Operations

The unary operators, as their name implies, require only a single argument.

For example, in the following code segment, the value of the variable "i" is incremented. This is a shorthand method of executing the operation "$i = i + 1$;"

```
unsigned int    i;

i++;
```

Example: It is not uncommon in embedded system design projects to have every pin on a microcontroller employed. Furthermore, it is not uncommon to have multiple inputs and outputs assigned to the same port but on different port input/output pins. Some compilers support specific pin reference. Another technique that is not compiler specific is **bit twiddling**. Figure 2.6 shows bit twiddling examples on how individual bits may be manipulated without affecting other bits using bitwise and unary operators. The information shown here was extracted from the ImageCraft ICC AVR compiler documentation (`www.imagecraft.com`).



Syntax	Description	Example
a | b	bitwise or	PORTA |= 0x80; // turn on bit 7 (msb)
a & b	bitwise and	if ((PINA & 0x81) == 0) // check bit 7 and bit 0
a ^ b	bitwise exclusive or	PORTA ^= 0x80; // flip bit 7
~a	bitwise complement	PORTA &= ~0x80; // turn off bit 7

Figure 2.6: Bit twiddling (`www.imagecraft.com`).

2.4.2 PROGRAMMING CONSTRUCTS

In this section, we discuss several methods of looping through a piece of code. We will examine the "for" and the "while" looping constructs.

The **for** loop provides a mechanism for looping through the same portion of code a fixed number of times. The for loop consists of three main parts:

- loop initiation,
- loop termination testing, and
- loop increment.

In the following code fragment, the for loop is executed ten times.

```
unsigned int  loop_ctr;

for(loop_ctr = 0; loop_ctr < 10; loop_ctr++)
  {
                          //loop body


  }
```

The for loop begins with the variable "loop_ctr" equal to 0. During the first pass through the loop, the variable retains this value. During the next pass through the loop, the variable "loop_ctr" is incremented by one. This action continues until the "loop_ctr" variable reaches the value of ten. Since the argument to continue the loop is no longer true, program execution continues after the close bracket for the for loop.

In the previous example, the for loop counter was incremented at the beginning of each loop pass. The "loop_ctr" variable can be updated by any amount. For example, in the following code fragment the "loop_ctr" variable is increased by three for every pass of the loop.

```
unsigned int  loop_ctr;

for(loop_ctr = 0; loop_ctr < 10; loop_ctr=loop_ctr+3)
  {
                          //loop body


  }
```

The "loop_ctr" variable may also be initialized at a high value and then decremented at the beginning of each pass of the loop.

```
unsigned int  loop_ctr;

for(loop_ctr = 10; loop_ctr > 0; loop_ctr--)
  {
                          //loop body


  }
```

As before, the "loop_ctr" variable may be decreased by any numerical value as appropriate for the application at hand.

The **while** loop is another programming construct that allows multiple passes through a portion of code. The while loop will continue to execute the statements within the open and close brackets while the condition at the beginning of the loop remains logically true. The code snapshot below will implement a ten iteration loop. Note how the "loop_ctr" variable is initialized outside of the loop and incremented within the body of the loop. As before, the variable may be initialized to a greater value and then decremented within the loop body.

```
unsigned int  loop_ctr;

loop_ctr = 0;
while(loop_ctr < 10)
  {
                         //loop body
  loop_ctr++;
  }
```

Frequently, within a microcontroller application, the program begins with system initialization actions. Once initialization activities are complete, the processor enters a continuous loop. This may be accomplished using the following code fragment:

```
while(1)
  {



  }
```

2.4.3 DECISION PROCESSING

There are a variety of constructs that allow decision making. These include the following:

- **if** statement,
- **if-else** construct,
- **if-else if-else** construct, and
- **switch** statement.

The **if** statement will execute the code between an open and close bracket set should the condition within the if statement be logically true.

Example: To help develop the algorithm for steering the Dagu Magician robot through a maze, an LED is connected to PORTB pin 1 on the ATmega164. The robot's center IR sensor is connected to an analog-to-digital converter at PORTC, pin 1. The IR sensor provides a voltage output that is inversely proportional to distance of the sensor from the maze wall. It is desired to illuminate the LED if the robot is within 10 cm of the maze wall. The sensor provides an output voltage of 2.5 VDC at the 10 cm range. The following **if** statement construct implements this LED indicator. We provide the actual code to do this later in the chapter.

```
if (PORTC[1] > 2.5)            //Center sensor voltage greater than 2.5V
   {
   PORTB = 0x02;               //illuminate LED on PORTB[1]
   }
```

In the example provided, there is no method to turn off the LED once it is turned on. This will require the **else** portion of the construct as shown in the next code fragment.

```
if (PORTC[1] > 2.5)            //Center sensor voltage greater than 2.5V
   {
   PORTB = 0x02;               //illuminate LED on PORTB[1]
   }
else
   {
   PORTB = 0x00;               //extinguish the LED on PORTB[1]
   }
```

The **if-else if-else** construct may be used to implement a three LED system. In this example, the left, center, and right IR sensors are connected to analog-to-digital converter channels on PORTC pins 2, 1, and 0, respectively. The LED indicators are connected to PORTB pins 2, 1, and 0. The following code fragment implements this LED system.

```
if (PORTC[2] > 2.5)            //Left sensor voltage greater than 2.5V
   {
   PORTB = 0x04;               //illuminate LED on PORTB[2]
   }
else if (PORTC[1] > 2.5)       //Center sensor voltage greater than 2.5V
   {
   PORTB = 0x02;               //illuminate the LED on PORTB[1]
   }
else if (PORTC[0] > 2.5)       //Right sensor voltage greater than 2.5V
   {
   PORTB = 0x01;               //illuminate the LED on PORTB[0]
   }
else                           //no walls sensed within 10 cm
   {
   PORTB = 0x00;               //extinguish LEDs
   }
```

The **switch** statement is used when multiple if-else conditions exist. Each possible condition is specified by a case statement. When a match is found between the switch variable and a specific case entry, the statements associated with the case are executed until a **break** statement is encountered.

Example: Suppose eight pushbutton switches are connected to PORTD. Each switch will implement a different action. A switch statement may be used to process the multiple possible decisions as shown in the following code fragment:

```
void read_new_input(void)
{
new_PORTD = PIND;

if(new_PORTD != old_PORTD)                 //check for status change PORTD

switch(new_PORTD)
  {                                        //process change in PORTD input
  case 0x01:                               //PD0
                                           //PD0 related actions
             break;

  case 0x02:                               //PD1
                                           //PD1 related actions
             break;

  case 0x04:                               //PD2
                                           //PD2 related actions
             break;

  case 0x08:                               //PD3
                                           //PD3 related actions
             break;

  case 0x10:                               //PD4
                                           //PD4 related actions
             break;

  case 0x20:                               //PD5
                                           //PD5 related actions
             break;

  case 0x40:                               //PD6
                                           //PD6 related actions
             break;
  case 0x80:                                    //PD7
                                                //PD7 related actions
             break;

             default:;                          //all other cases
      }                                         //end switch(new_PORTD)
  }                                             //end if new_PORTD
old_PORTD=new_PORTD;                            //update PORTD
}
```

That completes our brief overview of the C programming language.

2.5 APPLICATION

In this example, a for loop counts from 0–100. Within the body of the loop, the current count value is examined to determine which numbers evenly divide into count. The remainder operator (%) is used.

```
//***************************************************************
//file name: chp2_app
//function: provides test bench for MICROCHIP AVR ATmega164 controller
//target controller: MICROCHIP ATmega164
//
//MICROCHIP AVR ATmega164 Controller Pin Assignments
//Chip Port Function I/O Source/Dest Asserted Notes
//Pin 1 - 7: PB0 to PB7 set to output: unused
//Pin 9 Reset
//Pin 10 VDD
//Pin 11 Gnd
//Pin 12 and 13 Resonator
//Pin 14 - 21: PD0 to PD7 output to tristate LED indicator
//Pin 22 - 29: PC0 to PC7 set to output: unused
//Pin 30 AVcc to VDD
//Pin 31 AGnd to Ground
//Pin 32 ARef to Vcc
//Pin 33 - 40: PA7 to PA0 set to output: unused
//
//External resonator: 20 MHz
//
//author: Steven Barrett and Daniel Pack
//created: Aug 19, 2016
```

```
//last revised: Aug 19, 2016
//***************************************************************

//function prototypes********************************************
void initialize_ports(void);       //initializes ports
void delay_100ms(void);
void delay_1s(void);

//Include Files: choose the appropriate include file depending on
//the compiler in use - comment out the include file not in use.

//include file(s) for JumpStart C for AVR Compiler****************
#include<iom164pv.h>                   //contains reg definitions

//include file(s) for the Microchip Studio gcc compiler
//#include <avr/io.h>                  //contains reg definitions


int main(void)
{
unsigned char count = 0;

initialize_ports();

for(count = 0; count <=100; count++)
  {

  if(count
  if(count
  if(count
  if(count
  if(count
  if(count
  if(count
  if(count

  PORTD = 0x00;                  //turn off all LEDs

  delay_1s();
  }

}

//****************************************************************
//function definitions
//****************************************************************

//****************************************************************
//initialize_ports: provides initial configuration for I/O ports
//****************************************************************
void initialize_ports(void)
{
DDRA=0xff;                          //set PORTA[7:0] as output
PORTA=0x00;                         //initialize low

DDRB=0xff;                          //PORTB[7:0] as output
PORTB=0x00;                         //initialize low

DDRC=0xff;                          //set PORTC as output
PORTC=0x00;                         //initialize low

DDRD=0xff;                          //set PORTD as output
PORTD=0x00;                         //initialize low

}

//****************************************************************
//delay_100ms: inaccurate, yet simple method of creating delay
//   - processor clock: ceramic resonator at 20 MHz
//   - 100 ms delay requires 2M clock cycles
//   - nop requires 1 clock cycle to execute
//****************************************************************
void delay_100ms(void)
{
unsigned int i,j;

for(i=0; i < 2000; i++)
  {
```

```
  for(j=0; j < 1000; j++)
    {
    asm("nop");
    }
  }
}

//*************************************************************
//delay_1s: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 20 MHz
//  - 100 ms delay requires 2M clock cycles
//  - nop requires 1 clock cycle to execute
//  - call 10 times for 1s delay
//*************************************************************

void delay_1s(void)
{
unsigned int i;

for(i=0; i< 10; i++)
  {
  delay_100ms();
  }
}

//*************************************************************
```

2.6 LABORATORY EXERCISE

Write a program in C that consists of four functions.

- The first function displays an incrementing binary count from 0–255 on PORTD. PORTB[0] illuminates for even numbers and PORTB[1] illuminates for odd numbers.
- The second function displays a decrementing count on the PORTD. PORTB[0] illuminates numbers evenly divisible by 5 and PORTB[1] illuminates for numbers evenly divisible by 10.
- The third function displays an incrementing binary count from 0–255 followed by a decrementing count from 255–0 on PORTD. PORTB[0] illuminates when the count is greater than 128 and PORTB[1] illuminates for numbers less than 128. Both LEDs illuminate when the count equals 128.

- The fourth function is called delay_30ms. It should provide a delay of at least 30 ms.
- Call each of the first three functions sequentially from the main program. The 30 ms delay function should be called every time the PORTD is updated with a new value.

2.7 SUMMARY

The goal of this chapter was to provide a tutorial on how to begin programming. We used a top-down approach by starting with the "big picture" of writing a program followed by an overview of the major pieces of a typical program. We then discussed the basics of the C programming language. Examples were used to illustrate programming principles.

2.8 REFERENCES AND FURTHER READING

Barrett, S. F. and Pack, D. J. *Atmel AVR® Microcontroller Primer Programming and Interfacing*. Morgan & Claypool Publishers, 2008.

Barrett, S. F. and Pack, D. J. *Embedded Systems Design and Applications with the 68HC12 and HCS12*, Pearson Prentice Hall, 2005. 37, 38, 39

Barrett, S. F. *Embedded Systems Design with the Atmel AVR® Microcontroller*. Morgan & Claypool Publishers, 2010.

Barrett, S. F. and Pack, D. J. *Microcontrollers Fundamentals for Engineers and Scientists*. Morgan & Claypool Publishers, 2006. DOI: 10.2200/s00025ed1v01y200605dcs001.

ImageCraft embedded systems C development tools. Palo Alto, CA, 2019. www.imagecraft.com

2.9 CHAPTER PROBLEMS

1. Describe the key components of a C program.
2. What is an include file?
3. What are the three pieces of code required for a program function?
4. Describe how to define a program constant.
5. Provide the C program statement to set PORTB pins 1 and 7 to logic one. Use bit-twiddling techniques.
6. Provide the C program statement to reset PORTB pins 1 and 7 to logic zero. Use bit-twiddling techniques.

7. What is the difference between a for and while loop?

8. When should a switch statement be used vs. the if-then statement construct?

CHAPTER 3

Serial Communication Subsystem

Objectives: After reading this chapter, the reader should be able to:

- describe the differences between serial and parallel communication capabilities of microcontrollers;
- provide definitions for key serial communications terminology;
- describe the operation of the universal serial asynchronous receiver transmitter (USART);
- program the USART for basic transmission and reception;
- describe the operation of the serial peripheral interface (SPI);
- program the SPI for basic transmission and reception;
- describe the purpose of the two-wire interface (TWI); and
- program the TWI for basic write operation.

3.1 OVERVIEW

Microcontrollers must often exchange data with other microcontrollers or peripheral devices. Data may be exchanged by using parallel or serial techniques. With parallel techniques, an entire byte of data is typically sent simultaneously from the transmitting device to the receiver device. Although this is efficient from a time point of view, it requires eight separate lines for the data transfer [Barrett and Pack, 2006].

In serial transmission, on the other hand, a byte of data is sent a single bit at a time. Once 8 bits have been received at the receiver, the data byte may be used. Although this is inefficient from a time point of view, it only requires a line (or two) to transmit the data.

The ATmega164 is equipped with a host of different serial communication subsystems, including the serial USART, SPI, and TWI. What all of these systems have in common is the serial transmission of data. Before discussing the different serial communication features aboard the ATmega164, we review serial communication terminology.

3.2 SERIAL COMMUNICATION TERMINOLOGY

In this section, we review common terminology associated with serial communication.

3.2.1 ASYNCHRONOUS VS. SYNCHRONOUS SERIAL TRANSMISSION

In serial communications, the transmitting and receiving device must be synchronized to one another and use a common data rate and protocol. Synchronization allows both the transmitter and receiver to be expecting data transmission/reception at the same time. There are two basic methods of maintaining "sync" between the transmitter and receiver: asynchronous and synchronous.

In an asynchronous serial communication system, such as the USART aboard the ATmega164, framing bits are used at the beginning and end of a data byte. These framing bits alert the receiver that an incoming data byte has arrived and also signals the completion of the data byte reception. The data rate for an asynchronous serial system is typically much slower than the synchronous system, but it only requires a single wire and a common ground between the transmitter and receiver.

A synchronous serial communication system maintains "sync" between the transmitter and receiver by employing a common clock between the two devices. Data bits are sent and received on the common clock signal's rising or falling edge. This allows higher data transfer rates higher than with asynchronous techniques, but the synchronous technique requires two lines, data and clock, and a common ground to connect the receiver and transmitter.

3.2.2 BAUD RATE

Data transmission rates are typically specified as a baud or bits per second rate. For example, 9600 baud indicates data are being transferred at 9600 bits per second.

3.2.3 FULL DUPLEX

Often, serial communication systems must both transmit and receive data. To do both transmission and reception simultaneously requires separate hardware for transmission and reception. A single duplex system has a single complement of hardware that must be switched from transmission to reception configuration. A full duplex serial communication system has separate hardware for transmission and reception.

3.2.4 NONRETURN TO ZERO CODING FORMAT

There are many different coding standards used within serial communications. The important point is the transmitter and receiver must use a common coding standard so data may be interpreted correctly at the receiving end. The Microchip ATmega164 [Microchip] uses a nonreturn to zero coding standard. In nonreturn to zero, coding a logic one is signaled by a logic high during the entire time slot allocated for a single bit, whereas a logic zero is signaled by a logic low during the entire time slot allocated for a single bit.

3.2.5 THE RS-232 COMMUNICATION PROTOCOL

When serial transmission occurs over a long distance, additional techniques may be used to ensure data integrity. Over long distances, logic levels degrade and may be corrupted by noise. At the receiving end, it is difficult to discern a logic high from a logic low. The RS-232 standard has been around for some time. With the RS-232 standard (EIA-232), a logic one is represented with a -12 VDC level, whereas a logic zero is represented by a $+12$ VDC level. Chips are commonly available (e.g., MAX232) that convert the 5 and 0 VDC output levels from a transmitter to RS-232-compatible levels and convert back to 5 and 0 V levels at the receiver. The RS-232 standard also specifies other configuration features for this communication protocol.

3.2.6 PARITY

To further enhance data integrity during transmission, parity techniques may be used. Parity is an additional bit (or bits) that may be transmitted with the data byte. The ATmega164 uses a single parity bit. With a single parity bit, a single-bit error may be detected. Parity may be even or odd. In even parity, the parity bit is set to one or zero, such that the number of ones in the data byte including the parity bit is even. In odd parity, the parity bit is set to one or zero, such that the number of one's in the data byte including the parity bit is odd. At the receiver, the number of bits within a data byte including the parity bit are counted to ensure that parity has not changed, indicating an error, during transmission.

3.2.7 AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE

The American Standard Code for Information Interchange (ASCII) is a standardized 7-bit method of encoding alphanumeric data. It has been in use for many decades, so some of the characters and actions listed in the ASCII table are not in common use today. However, ASCII is still the most common method of encoding alphanumeric data. The ASCII code is shown in Figure 3.1. For example, the capital letter "G" is encoded in ASCII as 0x47. Recall the "0x" symbol indicates the hexadecimal number representation. Unicode is the international counterpart of ASCII. It provides standardized 16-bit encoding format for the written languages of the world. ASCII is a subset of Unicode. The interested reader is referred to the Unicode home page website at `www.unicode.org` for additional information on this standardized encoding format.

3.3 SERIAL USART

The serial USART subsystem provides full duplex (two-way) communication between a receiver and transmitter pair. This is accomplished by equipping the ATmega164 with two independent

		Most Significant Digit							
		0x0_	0x1_	0x2_	0x3_	0x4_	0x5_	0x6_	0x7_
Least Significant Digit	0x_0	NUL	DLE	SP	0	@	P	`	p
	0x_1	SOH	DC1	!	1	A	Q	a	q
	0x_2	STX	DC2	“	2	B	R	b	r
	0x_3	ETX	DC3	#	3	C	S	c	s
	0x_4	EOT	DC4	$	4	D	T	d	t
	0x_5	ENQ	NAK	%	5	E	U	e	u
	0x_6	ACK	SYN	&	6	F	V	f	v
	0x_7	BEL	ETB	‘	7	G	W	g	w
	0x_8	BS	CAN	(	8	H	X	h	x
	0x_9	HT	EM	)	9	I	Y	i	y
	0x_A	LF	SUB	*	:	J	Z	j	z
	0x_B	VT	ESC	+	;	K	[	k	{
	0x_C	FF	FS	‘	<	L	\	l	|
	0x_D	CR	GS	-	=	M	]	m	}
	0x_E	SO	RS	.	>	N	^	n	~
	0x_F	SI	US	/	?	O	_	o	DEL

Figure 3.1: ASCII code. The ASCII code is used to encode alphanumeric characters. The “0x” indicates hexadecimal notation in the C programming language.

hardware channels for the transmitter and receiver. Furthermore, the ATmega164 is equipped with two separate USART channels designated USART0 and USART1. The USART system is typically used for asynchronous communication. That is, there is not a common clock between the transmitter and receiver to keep them synchronized with one another. To maintain synchronization between the transmitter and receiver, framing start and stop bits are used at the beginning and end of each data byte in a transmission sequence. The Microchip USART also has synchronous features. The ATmega164 USART subsystem is quite flexible. It has the capability to be set to a variety of data transmission rates or baud (bits per second) rates. The USART may also be set for data bit widths of 5 to 9 bits with one or two stop bits. Furthermore, the ATmega164 is equipped with a hardware-generated parity bit (even or odd) and parity check hardware at the receiver. A single parity bit allows for the detection of a single bit error within a byte of data. The flexibility of settings allow the USART to be configured with a wide variety of USART compatible peripheral devices. We now discuss the operation, programming, and application of the USART. Because of space limitations, we cover only the most basic capability of this flexible and powerful serial communication system.

3.3.1 SYSTEM OVERVIEW

The block diagram for the USART is shown in Figure 3.2. The block diagram may appear a bit overwhelming, but realize that there are four basic pieces to the diagram: the clock generator, the transmission hardware, the receiver hardware, and three control registers (UCSRxA, UCSRxB, and UCSRxC). **Note:** The "x" in each register name should be designated as either "0" or "1" for the specific USART channel being used. We discuss each USART piece in turn.

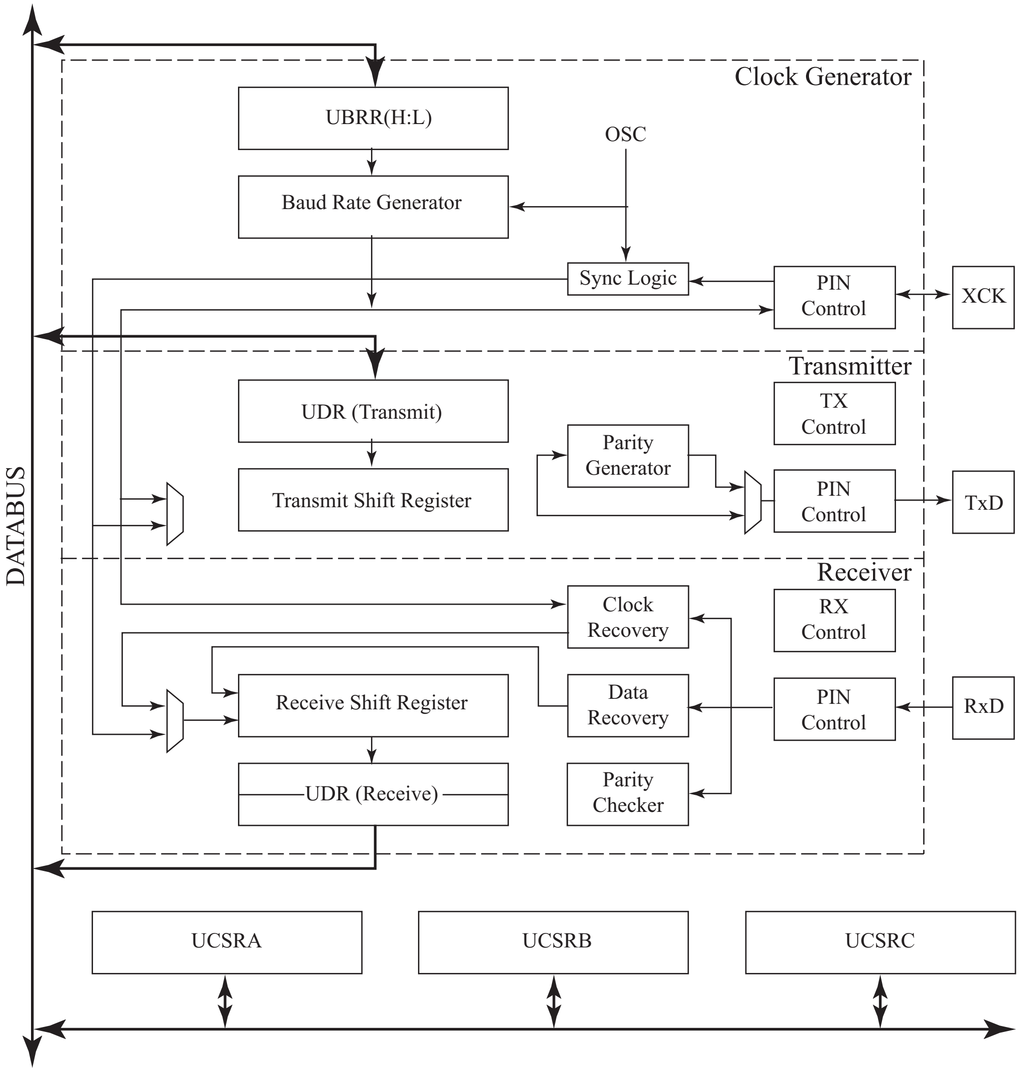


Figure 3.2: Microchip AVR ATmega164 USART block diagram. (Figure used with permission of Microchip Technology, Inc. (www.microchip.com). All rights reserved.)

USART Clock Generator

The USART Clock Generator provides the clock source for the USART system and sets the baud rate for the USART. The baud rate is derived from the overall microcontroller clock source. The overall system clock is divided by the USART baud rate registers UBRRx[H:L] and several additional dividers to set the desired baud rate. For the asynchronous normal mode (U2X bit = 0), the baud rate is determined using the following expression:

$$\text{baud rate} = (\text{system clock frequency})/(2(\text{UBRRx} + 1)),$$

where UBRRx is the content of the UBRRxH and UBRRxL registers (0–4095). Solving for UBRRx yields

$$\text{UBRxR} = ((\text{system clock generator})/(16 \times \text{baud rate})) - 1.$$

USART Transmitter

The USART transmitter consists of a Transmit Shift Register. The data to be transmitted are loaded into the Transmit Shift Register via the USART I/O data register (UDRx). The start and stop framing bits are automatically appended to the data within the Transmit Shift Register. The parity is automatically calculated and appended to the Transmit Shift Register. Data are then shifted out of the Transmit Shift Register via the TxD pin a single bit at a time at the established baud rate. The USART transmitter is equipped with two status flags: the USART data register empty (UDRE) and the transmit complete (TXC) flags. The UDRE flag sets when the transmit buffer is empty, indicating it is ready to receive new data. This bit should be written to a zero when writing the USART Control and Status Register A (UCSRxA). The UDRE bit is cleared by writing to the UDRx register. The TXC flag bit is set to logic 1 when the entire frame in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer. The TXC bit may be reset by writing a logic 1 to it.

USART Receiver

The USART receiver is virtually identical to the USART transmitter except for the direction of the data flow, which is reversed. Data are received a single bit at a time via the RxD pin at the established baud rate. The USART receiver is equipped with the receive complete (RXC) flag. The RXC flag is logic 1 when unread data exist in the receive buffer.

USART Registers

In this section, we discuss the register settings for controlling the USART system. We have already discussed the function of the UDRx and the USART baud rate registers (UBRRxH and UBRRxL). As a friendly reminder, the “x” in each register name should be designated as either “0” or “1” for the specific USART channel being used. **Note:** The USART Control and Status Register C (UCSRxC) and the USART baud rate register high (UBRRxH) are assigned to the same I/O location in the memory map (Figure 3.3). The URSEL bit (bit 7 of both registers)

USART Control and Status Register A (UCSRxA)

RXCx	TXCx	UDREx	FEx	DORx	UPEx	U2Xx	MPCMx
7							0

USART Control and Status Register B (UCSRxB)

RXCIEx	TXCIEx	UDRIEx	RXENx	TXENx	UCSZx2	RXB8x	TXB8x
7							0

USART Control and Status Register C (UCSRxC)

UMSELx1	UMSELx0	UPMx1	UPMx0	USBSx	UCSZx1	UCSZx0	UCPOLx
7							0

USART Data Register - UDRx

UDRx(Read)

RXB7	RXB6	RXB5	RXB4	RXB3	RXB2	RXB1	RXB0

UDRx(Write)

TXB7	TXB6	TXB5	TXB4	TXB3	TXB2	TXB1	TXB0
7							0

USART Baud Rate Registers - UBRRxH and UBRRxL

UBRRxH

URSEL=0	---	---	---	UBRR11	UBRR10	UBRR9	UBRR8

UBRRxL

UBRR7	UBRR6	UBRR5	UBRR4	UBRR3	UBRR2	UBRR1	UBRR0
7							0

Note: The "x" must be set to a "0" or "1" for the specific USART channel being used.

Figure 3.3: USART registers.

determines which register is being accessed. The URSEL bit must be 1 when writing to the UCSRxC register and 0 when writing to the UBRRxH register.

UCSRxA. This register contains the RXCx, TXCx, and the UDREx bits. The function of these bits has already been discussed.

UCSRxB. This register contains the receiver and transmitter enable bits (RXENx and TXENx, respectively). These bits are the "on/off" switch for the receiver and transmitter, respectively. The UCSRxB register also contains the UCSZx2 bit. The UCSZx2 bit in the UC-

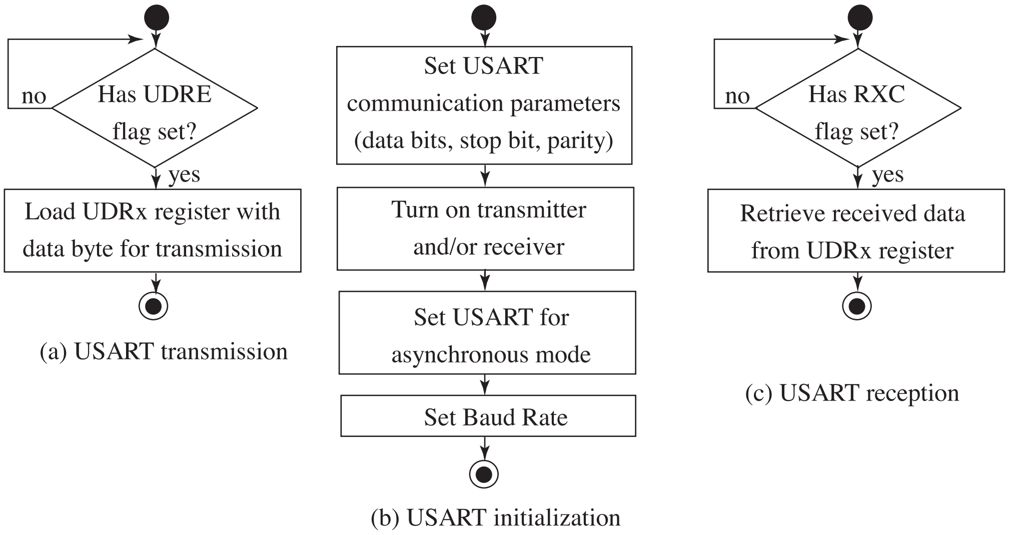
SRxB register and the UCSZx[1:0] bits contained in the UCSRxC register together set the data character size.

UCSRxC. This allows the user to customize the data features to the application at hand. It should be emphasized that both the transmitter and receiver be configured with the same data features for proper data transmission. The UCSRxC contains the following bits:

- USART mode select (UMSELx[1:0]): 0, asynchronous USART operation; 1, synchronous USART operation;
- USART parity mode (UPMx[1:0]): 00, no parity; 10, even parity; 11, odd parity;
- USART stop bit select (USBSx): 0, one stop bit; 1, two stop bits; and
- USART character size (data width) (UCSZx[2:0]): 000, 5 bits; 001, 6 bits; 010; 7 bits; 011, 8 bits; 111, 9 bits

3.3.2 SYSTEM OPERATION AND PROGRAMMING

The basic activities of the USART system consist of initialization, transmission, and reception. These activities are summarized in Figure 3.4. Both the transmitter and receiver must be initialized with the same communication parameters for proper data transmission. The transmission and reception activities are similar except for the direction of data flow. In transmission, we monitor for the UDRE flag to set, indicating the data register is empty. We then load the data


no
Has UDRE flag set?
yes
Load UDRx register with data byte for transmission
Set USART communication parameters (data bits, stop bit, parity)
Turn on transmitter and/or receiver
Set USART for asynchronous mode
Set Baud Rate
no
Has RXC flag set?
yes
Retrieve received data from UDRx register

(a) USART transmission

(b) USART initialization

(c) USART reception

Figure 3.4: USART activities.

for transmission into the UDRx register. For reception, we monitor for the RXC bit to set, indicating there are unread data in the UDRx register. We then retrieve the data from the UDRx register.

To program the USART, we implement the flow diagrams provided in Figure 3.4. In the sample code provided, we assume the ATmega164 is operating at 20 MHz and we desire a baud rate of 9600, asynchronous operation, no parity, one stop bit, and eight data bits on USART channel 1.

To achieve 9600 baud with an operating frequency of 20 MHz requires that we set the UBRR1 registers to 129, which is 0x81.

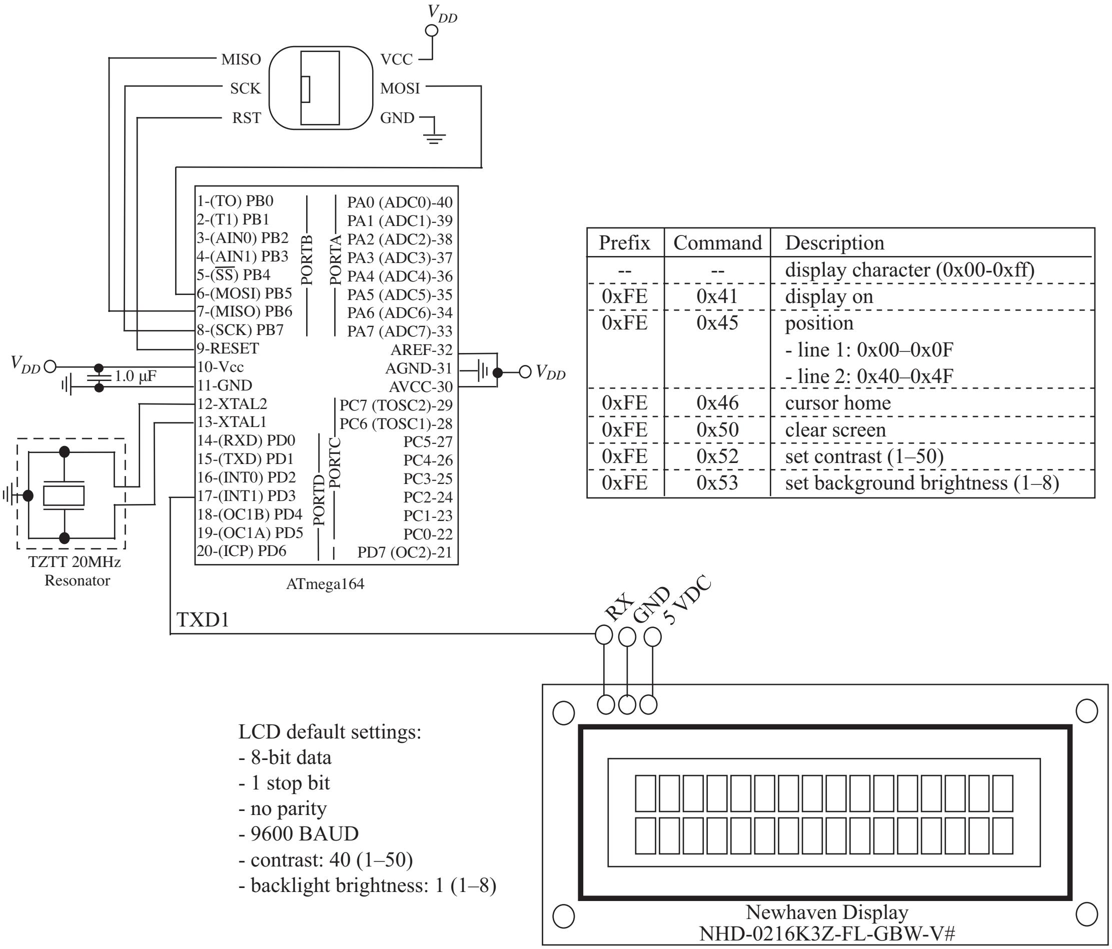
```
//***************************************************************
//USART_init: initializes the USART system
//Note: ATmega164 clock source 20 MHz ceramic resonator
//***************************************************************

void USART_init(void)
{
UCSR1A = 0x00;                //control register initialization
UCSR1B = 0x08;                //enable transmitter
UCSR1C = 0x86;                //async, no parity, 1 stop bit,
                              //8 data bits
                              //Baud Rate initialization
UBRR1H = 0x00; UBRR1L = 0x81;
}


//***************************************************************
//USART_transmit: transmits single byte of data
//***************************************************************

void USART_transmit(unsigned char data)
{
while((UCSR1A & 0x20)==0x00) //wait for UDRE flag
  {
  ;
  }
UDR1 = data;                  //load data to UDR1 for transmission
}

//***************************************************************
```

```
//USART_receive: receives single byte of data
//*************************************************************

unsigned char USART_receive(void)
{
while((UCSR1A & 0x80)==0x00) //wait for RXC flag
  {
  ;
  }
data = UDR1;                  //retrieve data from UDR1 return data
}

//*************************************************************
```

3.3.3 EXAMPLE: SERIAL LCD

When developing embedded solutions, it is useful to receive status information from the microcontroller. Often liquid crystal displays (LCDs) are used for status display. LCDs are available in serial or parallel configuration. Serial LCDs communicate with the microcontroller via the USART system. In this example we configure the Newhaven Display #NHD-0216K3Z-FL--GBW-V3 to communicate with the ATmega164. The interface is shown in Figure 3.5. ATmega164 USART channel 1 is used. An abbreviated command set for the LCD is also shown. Characters are sent directly to the LCD; commands must be preceded by 0xFE.

```
//*************************************************************
//serial_LCD
//*************************************************************

//Include Files: choose the appropriate include file depending on
//the compiler in use - comment out the include file not in use.

//include file(s) for JumpStart C for AVR Compiler****************
#include<iom164pv.h>                   //contains reg definitions

//include file(s) for the Atmel Studio gcc compiler
//#include <avr/io.h>                  //contains reg definitions

//function prototypes
void USART_init(void);
void USART_transmit(unsigned char data);
void LCD_init(void);
```



Prefix	Command	Description
--	--	display character (0x00-0xff)
0xFE	0x41	display on
0xFE	0x45	position - line 1: 0x00–0x0F - line 2: 0x40–0x4F
0xFE	0x46	cursor home
0xFE	0x50	clear screen
0xFE	0x52	set contrast (1–50)
0xFE	0x53	set background brightness (1–8)

LCD default settings:
- 8-bit data
- 1 stop bit
- no parity
- 9600 BAUD
- contrast: 40 (1–50)
- backlight brightness: 1 (1–8)

Figure 3.5: Serial LCD connections.

```
void lcd_print_string(char str[]);
void move_LCD_cursor(unsigned char position);

int main(void)
{
unsigned int i;

USART_init();
LCD_init();
for(i=0; i<10; i++)
  {
  USART_transmit('G');
  }

//move cursor to line 2, position 0
move_LCD_cursor(0x40);

lcd_print_string("Test 1 - 2");

}

//*************************************************************
//USART_init: initializes the USART system
//*************************************************************
void USART_init(void)
{
UCSR1A = 0x00; //control register initialization
UCSR1B = 0x08; //enable transmitter
UCSR1C = 0x86; //async, no parity, 1 stop bit,
               //8 data bits
               //Baud Rate initialization
UBRR1H = 0x00; UBRR1L = 0x81;
}

//*************************************************************
//USART_transmit: transmits single byte of data
//*************************************************************
void USART_transmit(unsigned char data)
{
while((UCSR1A & 0x20)==0x00) //wait for UDRE flag
  {
  ;
  }
UDR1 = data; //load data to UDR1 for transmission
}

//*************************************************************
//LCD_init: initializes the USART system
```

```
//*************************************************************
void LCD_init(void)
{
USART_transmit(0xFE);
USART_transmit(0x41);              //LCD on

USART_transmit(0xFE);
USART_transmit(0x46);              //cursor to home
}

//*************************************************************
//void lcd_print_string(char str[])
//*************************************************************

void lcd_print_string(char str[])
{
int k = 0;

while(str[k] != 0x00)
  {
  USART_transmit(str[k]);
  k = k+1;
  }
}

//*************************************************************
//void move_LCD_cursor(unsigned char position)
//*************************************************************

void move_LCD_cursor(unsigned char position)
{
USART_transmit(0xFE);
USART_transmit(0x45);
USART_transmit(position);
}

//*************************************************************
```

3.3.4 EXAMPLE: VOICE CHIP

Example: Voice chip. For speech synthesis, we use the SP0-512 text-to-speech chip (www.speechchips.com). The SP0-512 accepts UART compatible serial text stream. The text stream should be terminated with the carriage return control sequence (back slash r). The speech chip uses this symbol to indicate an end of transmission. The text stream is converted to phoneme codes used to generate an audio output. The chip requires a 9600 Baud bit stream with no parity, 8 data bits and a stop bit. The associated circuit is provided in Figure 3.6. Since the output from the ATmega164 is 5 VDC TTL compatible, it must be down converted to 3.3 VDC signal for the SP0-512 speech chip. Additional information on the chip and its features are available at www.speechchips.com. In the code below we use an exclamation mark to terminate the string of commands sent from the ATmega164. USART channel 1 is used to send the control signals to the SP0-512 speech chip.

```
//***************************************************************
//speech
//***************************************************************

//Include Files: choose the appropriate include file depending on
//the compiler in use - comment out the include file not in use.

//include file(s) for JumpStart C for AVR Compiler***************
#include<iom164pv.h>                   //contains reg definitions

//include file(s) for the Atmel Studio gcc compiler
//#include <avr/io.h>                  //contains reg definitions

//function prototypes
void USART_init(void);
void USART_transmit(unsigned char data);
void speak_string(char str[]);
void delay_100ms(void);
void delay_1s(void);

int main(void)
{
int i;

USART_init();
delay_1s();
```

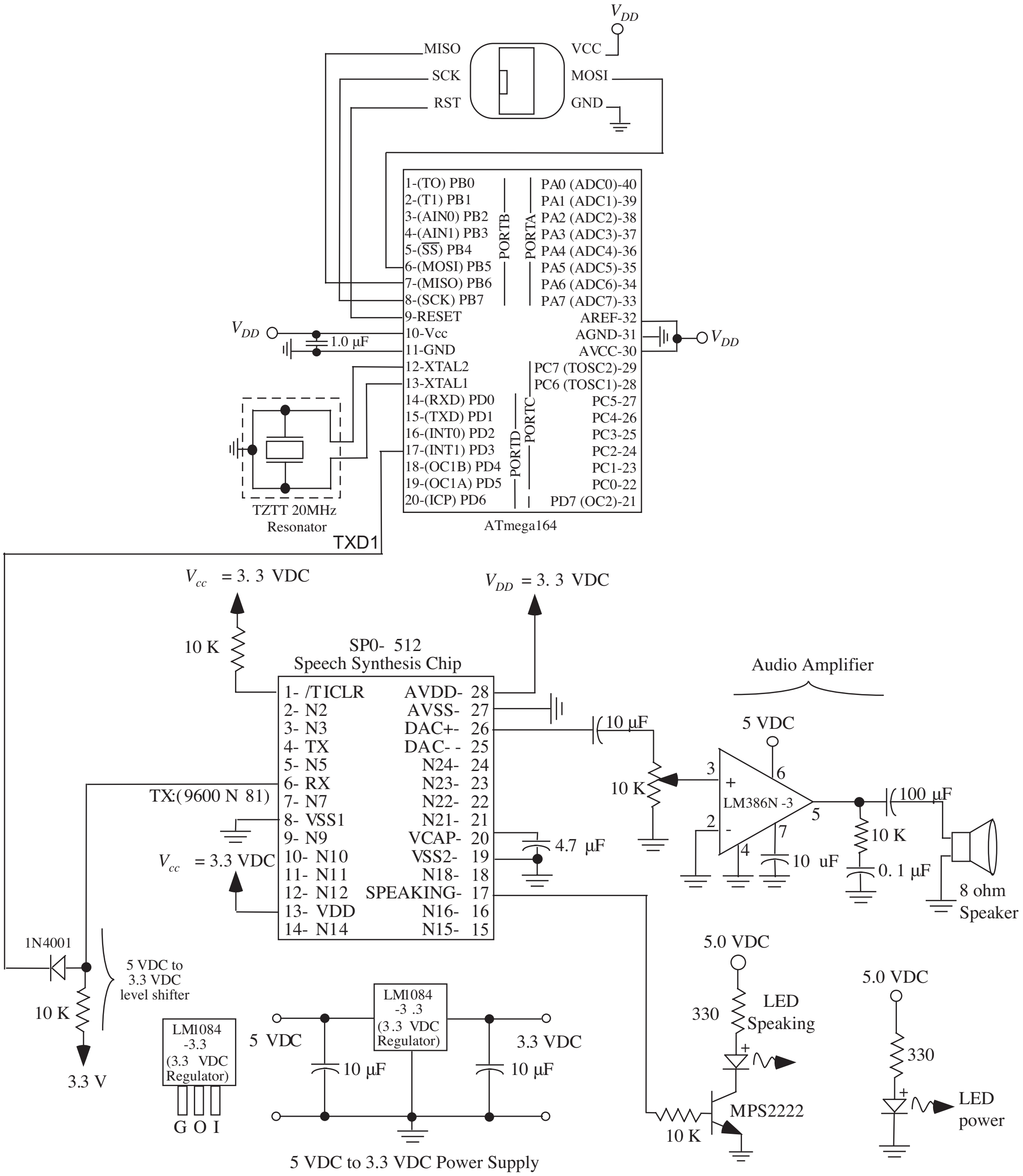


Figure 3.6: Speech synthesis support circuit (`www.speechchips.com`).

```
for(i=0; i<=4; i++) //speak five times
  {
  //terminate string with ! to signal end of TX
  speak_string("[BD]This [BD]is [BD]a [BD]test\r!");

  //delays
  delay_100ms();   //wait for speaking line to go high
  delay_100ms();   //wait for speaking line to go low
  delay_1s();      //pause between transmissions
  }
}

//*************************************************************
//USART_init: initializes the USART system
//*************************************************************
void USART_init(void)
{
UCSR1A = 0x00; //control register initialization
UCSR1B = 0x08; //enable transmitter
UCSR1C = 0x86; //async, no parity, 1 stop bit,
               //8 data bits
               //Baud Rate initialization
UBRR1H = 0x00; UBRR1L = 0x81;
}

//*************************************************************
//USART_transmit: transmits single byte of data
//*************************************************************
void USART_transmit(unsigned char data)
{
while((UCSR1A & 0x20)==0x00) //wait for UDRE flag
  {
  ;
  }
UDR1 = data; //load data to UDR1 for transmission
}

//*************************************************************
//void spring_string(char str[])
```

```
//*************************************************************

void speak_string(char str[])
{
int k = 0;

while(str[k] != 0x21)         //send characters until !
  {
  USART_transmit(str[k]);
  k = k+1;
  }
}

//***************************************************************
//delay_100ms: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 20 MHz
//  - 100 ms delay requires 2M clock cycles
//  - nop requires 1 clock cycle to execute
//***************************************************************
void delay_100ms(void)
{
unsigned int i,j;

for(i=0; i < 2000; i++)
  {
  for(j=0; j < 1000; j++)
    {
    asm("nop");
    }
  }
}

//***************************************************************
//delay_1s: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 20 MHz
//  - 100 ms delay requires 2M clock cycles
//  - nop requires 1 clock cycle to execute
//  - call 10 times for 1s delay
//***************************************************************
```

```
void delay_1s(void)
{
unsigned int i;

for(i=0; i< 10; i++)
  {
  delay_100ms();
  }
}

//*****************************************************************
```

3.3.5 EXAMPLE: PC SERIAL MONITOR

During embedded system development, it is helpful to receive viewable status back from the microcontroller. Limited status may be sent to an LCD. In this example, we provide a one-way link between the ATmega164 and a support computer (PC or laptop). This allows considerable status to be sent and displayed on the support computer's monitor.

The signal from the microcontroller is 5 VDC (if a 5 VDC power supply is used). For proper interface to the PC, the 5 VDC signal must be translated to a compatible PC signal. This is easily accomplished using a USB cable with FTDI (Future Technology Devices International—www.ftdichip.com) set to 5 VDC. This cable is available from a number of sources. We use a Sparkfun Electronics (www.sparkfun.com) DEV-09718 illustrated in Figure 3.7. Driver installation instructions for the cable is provided at the Sparkfun website.

Messages sent from the ATmega164 are displayed on the support computer's monitor using a serial monitor program. In this example the open-source Arduino Software integrated development environment (IDE) is used.

Provided below is a program illustrating how to send characters or messages from the ATmega164 to the support computer.

```
//***************************************************************
//usart_to_pc
//***************************************************************

//Include Files: choose the appropriate include file depending on
//the compiler in use - comment out the include file not in use.

//include file(s) for JumpStart C for AVR Compiler****************
#include<iom164pv.h>                   //contains reg definitions
```

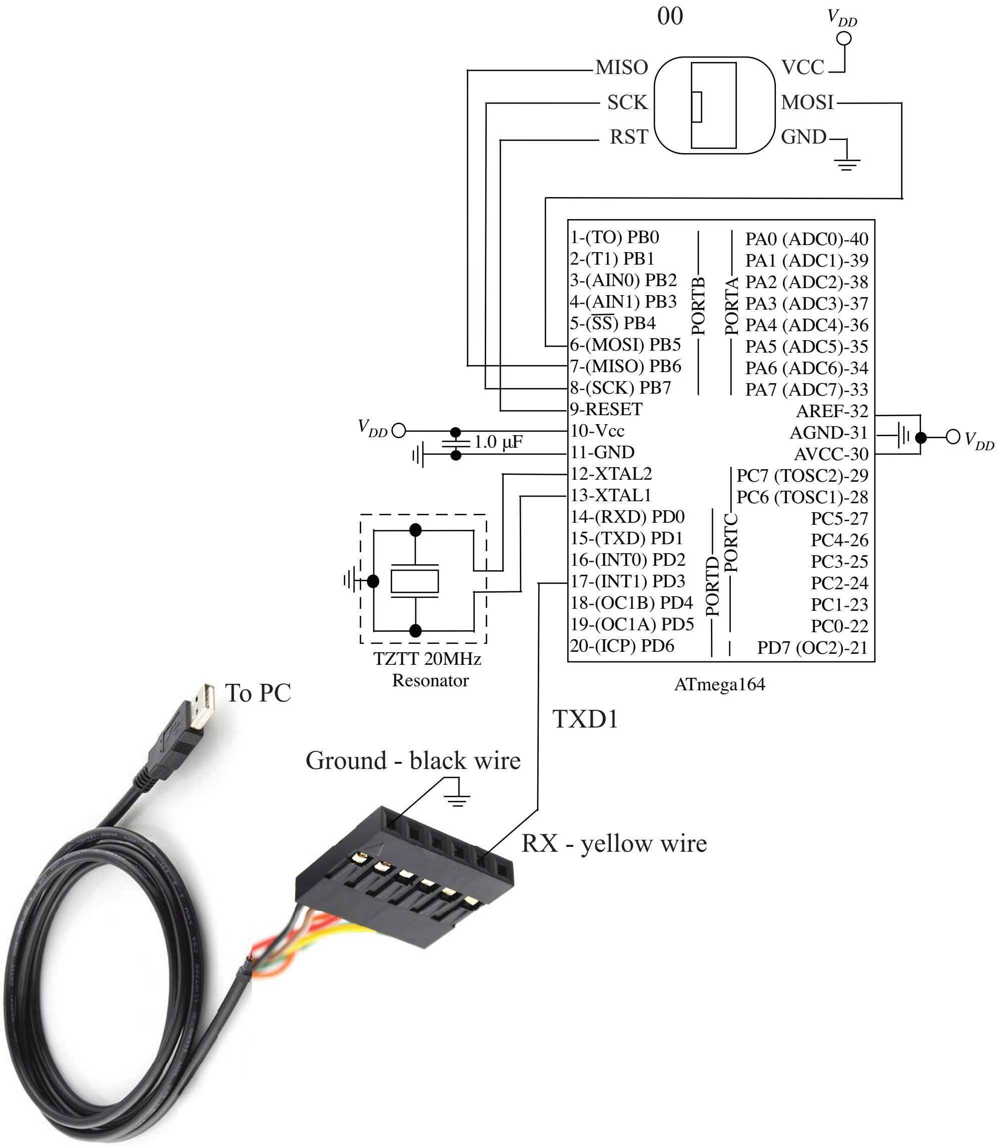


Figure 3.7: USART to computer communication link (`www.sparkfun.com`).

```
//include file(s) for the Atmel Studio gcc compiler
//#include <avr/io.h>                  //contains reg definitions

//function prototypes
void USART_init(void);
void USART_transmit(unsigned char data);
void lcd_print_string(char str[]);
void delay_100ms(void);
void delay_1s(void);


int main(void)
{
unsigned int i;

USART_init();

for(i=0; i<5; i++)
  {
  USART_transmit('G');
  lcd_print_string("\n");
  delay_100ms();
  }

for(i=0; i<5; i++)
  {
  lcd_print_string("Test print\n");
  delay_100ms();
  }

lcd_print_string("Test print\n\n");  //newline
lcd_print_string("Test \tprint\n");  //horizontal tab
}

//*************************************************************
//USART_init: initializes the USART system
//*************************************************************
void USART_init(void)
```

```
{
UCSR1A = 0x00; //control register initialization
UCSR1B = 0x08; //enable transmitter
UCSR1C = 0x86; //async, no parity, 1 stop bit,
           //8 data bits
           //Baud Rate initialization
UBRR1H = 0x00; UBRR1L = 0x81;
}

//*************************************************************
//USART_transmit: transmits single byte of data
//*************************************************************
void USART_transmit(unsigned char data)
{
while((UCSR1A & 0x20)==0x00) //wait for UDRE flag
  {
  ;
  }
UDR1 = data; //load data to UDR1 for transmission
}


//*************************************************************
//void lcd_print_string(char str[])
//*************************************************************

void lcd_print_string(char str[])
{
int k = 0;

while(str[k] != 0x00)
  {
  USART_transmit(str[k]);
  k = k+1;
  }
}

//***************************************************************
//delay_100ms: inaccurate, yet simple method of creating delay
```

```
//  - processor clock: ceramic resonator at 20 MHz
//  - 100 ms delay requires 2M clock cycles
//  - nop requires 1 clock cycle to execute
//***************************************************************
void delay_100ms(void)
{
unsigned int i,j;

for(i=0; i < 2000; i++)
  {
  for(j=0; j < 1000; j++)
    {
    asm("nop");
    }
  }
}

//***************************************************************
//delay_1s: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 20 MHz
//  - 100 ms delay requires 2M clock cycles
//  - nop requires 1 clock cycle to execute
//  - call 10 times for 1s delay
//***************************************************************

void delay_1s(void)
{
unsigned int i;

for(i=0; i< 10; i++)
  {
  delay_100ms();
  }
}

//***************************************************************
```

3.3.6 EXAMPLE: GLOBAL POSITIONING SYSTEM

The global positioning system (GPS) consists of a department of defense (DoD) constellation of 24 satellites in a 12,000 mile orbit. The system uses the concept of triangulation to locate a position on the earth. A GPS receiver requires reception from three satellites to obtain a two--dimensional fix (latitude and longitude), whereas four satellites are required to obtain a three-dimensional fix (latitude, longitude, and altitude). The accuracy of the GPS system is approximately 15 m; however, differential GPS or wide area augmentation system (WAAS) GPS provides accuracy to 3 m. The GPS system transmits over two different bands designated L1 and L2. The L1 civilian band transmits at 1575.42 MHz (ultra-high frequency or UHF) band. The signal may be received using low cost, microcontroller compatible receivers (see `www.sparkfun.com`). The GPS signal consists of a serial bit stream of 1500 bits. The signal may be transmitted in various formats. One format employs the National Marine Electronics Association NMEA-0183 format. This format is RS-232 compatible and transmits ASCII characters at a Baud rate of 4800 bits per second (BPS) using no parity and one stop bit. The serial bit stream contains satellite identification information, satellite health data, current date and time and location information. This information may be parsed from the bit stream [Garmin].

3.3.7 SERIAL PERIPHERAL INTERFACE

The ATmega164 SPI also provides for two-way serial communication between a transmitter and a receiver. In the SPI system, the transmitter and receiver share a common clock source. This requires an additional clock line between the transmitter and receiver but allows for higher data transmission rates as compared with the USART. The SPI system allows for fast and efficient data exchange between microcontrollers or peripheral devices. There are many SPI-compatible external systems available to extend the features of the microcontroller. For example, a liquid crystal display (LCD) or a digital-to-analog converter (DAC) could be added to the microcontroller using the SPI system.

SPI Operation

The SPI may be viewed as a synchronous 16-bit shift register with an 8-bit half residing in the transmitter and the other 8-bit half residing in the receiver as shown in Figure 3.8. The transmitter is designated as the master because it provides the synchronizing clock source between the transmitter and the receiver. The receiver is designated as the slave. A slave is chosen for reception by taking its slave select ($\overline{\text{SS}}$) line low. When the $\overline{\text{SS}}$ line is taken low, the slave's shifting capability is enabled. SPI transmission is initiated by loading a data byte into the master configured SPI data register (SPDR). At that time, the SPI clock generator provides clock pulses to the master and also to the slave via the SCK pin. A single bit is shifted out of the master designated shift register on the master out slave in (MOSI) microcontroller pin on every SCK pulse. The data are received at the MOSI pin of the slave designated device. At the same time, a single bit is shifted out of the master in slave out (MISO) pin of the slave device and into the

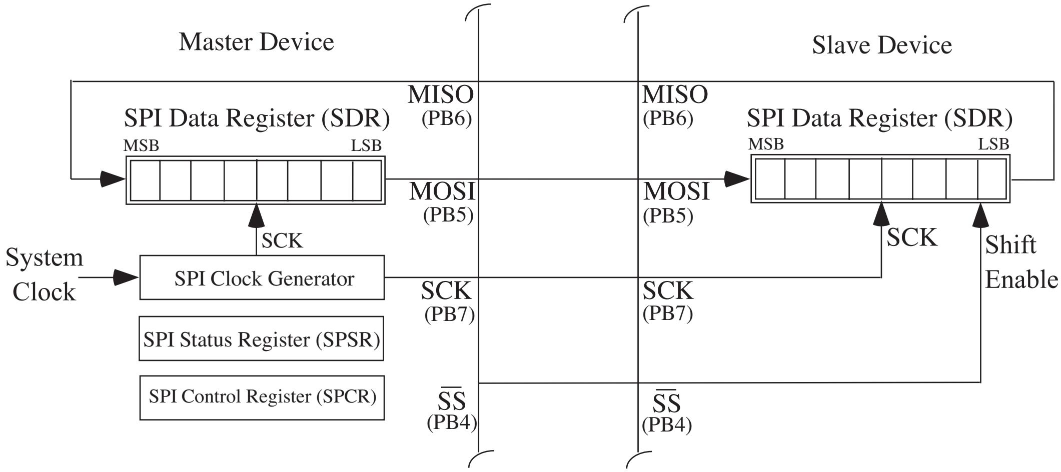


Figure 3.8: SPI overview.

MISO pin of the master device. After eight master SCK clock pulses, a byte of data has been exchanged between the master and slave designated SPI devices. Completion of data transmission in the master and data reception in the slave is signaled by the SPI interrupt flag (SPIF) in both devices. The SPIF flag is located in the SPI status register (SPSR) of each device. At that time, another data byte may be transmitted.

Registers

The registers for the SPI system are shown in Figure 3.9. We discuss each one in turn.

SPI Control Register - SPCR

SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
7							0

SPI Status Register - SPSR

SPIF	WCOL	---	---	---	---	---	SPI2X
7							0

SPI Data Register - SPDR

MSB							LSB
7							0

Figure 3.9: SPI registers.

SPI Control Register. The SPI Control Register (SPCR) contains the "on/off" switch for the SPI system. It also provides the flexibility for the SPI to be connected to a wide variety of devices with different data formats. It is important that both the SPI master and slave devices be configured for compatible data formats for proper data transmission. The SPCR contains the following bits.

- SPI enable (SPE) is the "on/off" switch for the SPI system. A logic 1 turns the system on and logic 0 turns it off.
- Data order (DORD) allows the direction of shift from master to slave to be controlled. When the DORD bit is set to 1, the least significant bit (LSB) of the SPDR is transmitted first. When the DORD bit is set to 0, the most significant bit (MSB) of the SPDR is transmitted first.
- The Master/Slave Select (MSTR) bit determines if the SPI system will serve as a master (logic 1) or slave (logic 0).
- The clock polarity (CPOL) bit determines the idle condition of the SCK pin. When CPOL is 1, SCK idles logic high, whereas when CPOL is 0, SCK idles logic 0.
- The clock phase (CPHA) determines if the data bit will be sampled on the leading (0) or trailing (1) edge of the SCK.
- The SPI SCK is derived from the microcontroller's system clock source. The system clock is divided down to form the SPI SCK. The SPI Clock Rate Select (SPR[1:0]) bits and the Double SPI Speed (SPI2X) bit are used to set the division factor. The following divisions may be selected using SPI2X, SPR1, and SPR0 bits:
  - 000: SCK = system clock/4
  - 001: SCK = system clock/16
  - 010: SCK = system clock/64
  - 011: SCK = system clock/128
  - 100: SCK = system clock/2
  - 101: SCK = system clock/8
  - 110: SCK = system clock/32
  - 111: SCK = system clock/64

SPI Status Register. This contains the SPIF. The flag sets when eight data bits have been transferred from the master to the slave. The SPIF bit is cleared by first reading the SPSR after the SPIF flag has been set and then reading the SPDR. The SPSR also contains the SPI2X bit used to set the SCK frequency.

SPI Data Register. As previously mentioned, writing a data byte to the SPDR initiates SPI transmission.

Programming

To program the SPI system, the system must first be initialized with the desired data format. Data transmission may then commence. Functions for initialization, transmission, and reception are provided below. In this specific example, we divide the clock oscillator frequency by 128 to set the SCK clock frequency.

```
//***************************************************************
//spi_init: initializes spi system
//***************************************************************

void spi_init(unsigned char control)
{
DDRB = 0xA0;           //Set SCK (PB7), MOSI (PB5) for output,
                       //others to input
                       //Configure SPI Control Register (SPCR)
SPCR = 0x53;           //SPIE:0,SPE:1,DORD:0,MSTR:1,CPOL:0,CPHA:0,
                       //
SPR:1,SPR0:1
}

//***************************************************************
//spi_write: Used by SPI master to transmit a data byte
//***************************************************************

void spi_write(unsigned char byte)
{
SPDR = byte; while (!(SPSR & 0x80));
}

//***************************************************************
//spi_read: Used by SPI slave to receive data byte
//***************************************************************

unsigned char spi_read(void)
{
while (!(SPSR & 0x80));
return SPDR;
}

//*************************************************************
```

3.3.8 EXAMPLE: ATMEGA164 PROGRAMMING

As discussed in Chapter 1, the ATmega164 may be programmed using a variety of techniques. We have chosen to use the Microchip AVR Dragon using the ISP programming technique. The ISP link between the host computer and the ATmega164 is a SPI connection.

3.3.9 EXAMPLE: LED STRIP

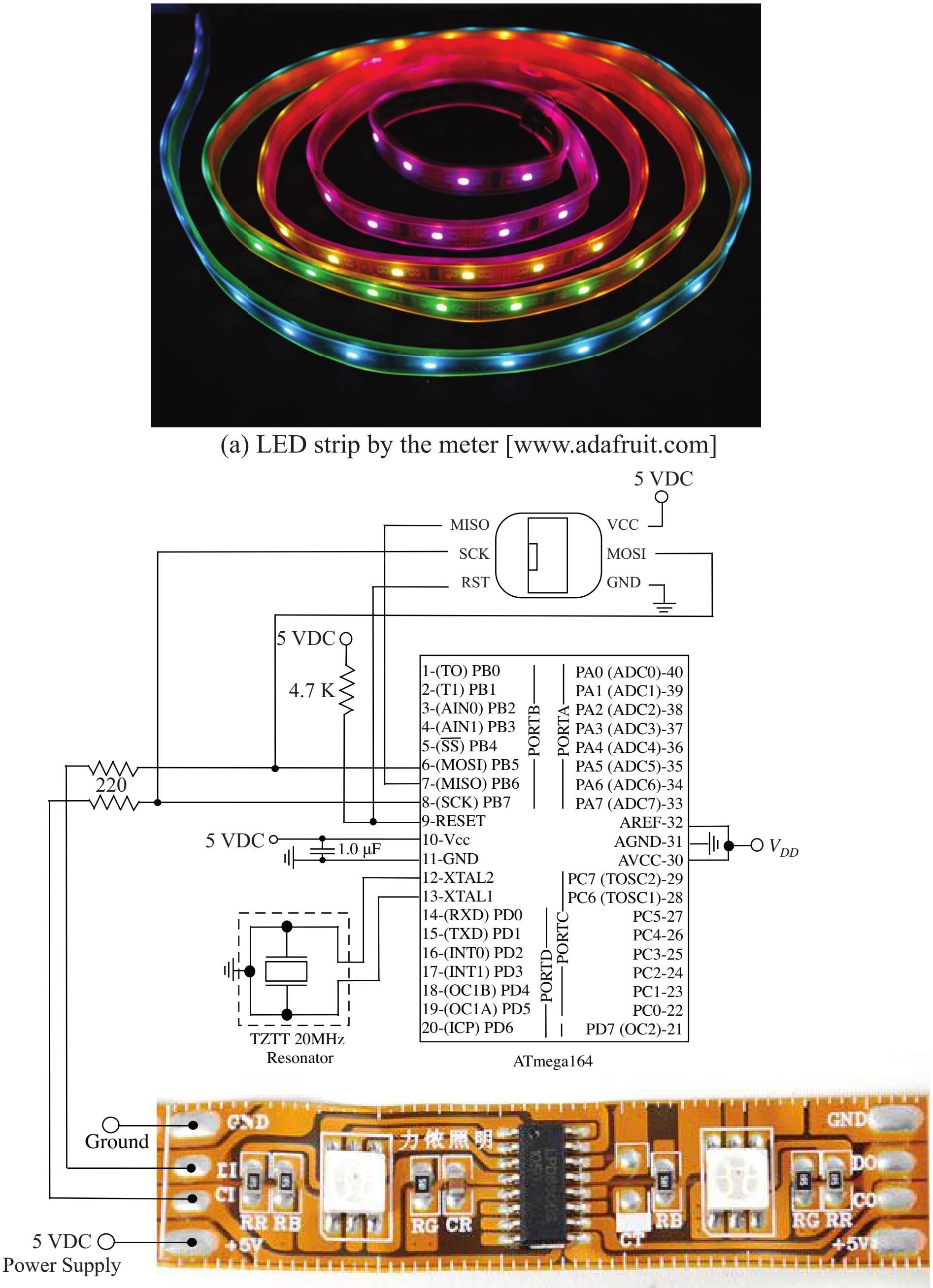
LED strips may be used for motivational (fun) optical displays, games, or for instrumentation-based applications. In this example we control an LPD8806-based LED strip. We use a one meter, 32 RGB LED strip available from Adafruit (#306) for approximately $30 USD (www.adafruit.com).

The red, blue, and green component of each RGB LED is independently set using an 8-bit code. The most significant bit (MSB) is logic one followed by 7 bits to set the LED intensity (0–127). The component values are sequentially shifted out of the ATmega164 using the serial peripheral interface (SPI) features. The first component value shifted out corresponds to the LED nearest the microcontroller. Each shifted component value is latched to the corresponding R, G, and B component of the LED. As a new component value is received, the previous value is latched and held constant. An extra byte is required to latch the final parameter value. A zero byte $(00)_{16}$ is used to complete the data sequence and reset back to the first LED (www.adafruit.com).

Only four connections are required between the ATmega164 and the LED strip as shown in Figure 3.10. The connections are color coded: red-power, black-ground, yellow-data, and green-clock. The ATmega164 is equipped with a single SPI channel. This channel is used to program the ATmega164 using ISP techniques. Additional series resistors on the MOSI and SCK lines are required to allow the single SPI channel to be also used to control the LED strip. It is important to note the LED strip requires a supply of 5 VDC and a current rating of 2 amps per meter of LED strip. In this example we use the Adafruit #276 5V 2A (2000 mA) switching power supply (www.adafruit.com).

In this example each RGB component is sent separately to the strip. The example illustrates how each variable in the program controls a specific aspect of the LED strip. Here are some important implementation notes.

- SPI must be configured for most significant bit (MSB) first.
- LED brightness is seven bits. Most significant bit (MSB) must be set to logic one.
- Each LED requires a separate R-G-B intensity component. The order of data is G-R-B.



(a) LED strip by the meter [www.adafruit.com]

(b) MSP432-EXP432P401R to LED strip connection [www.adafruit.com]

Figure 3.10: ATmega164 controlling LED strip (www.adafruit.com).

- After sending data for all LEDs. A byte of (0x00) must be sent to return strip to first LED.
- Data stream for each LED is: 1-G6-G5-G4-G3-G2-G1-G0-1-R6-R5-R4-R3-R2-R1-R0-1-B6-B5-B4-B3-B2-B1-B0.

```
//****************************************************************
//*******************************************************************
//spi.c
//*******************************************************************
//RGB_led_strip_tutorial: illustrates different variables within
//RGB LED strip
//
//LED strip LDP8806 - available from www.adafruit.com (#306)
//
//Connections:
// - External 5 VDC supply - Adafruit 5 VDC, 2A (#276) - red
// - Ground - black
// - Serial Data In:  ATmega164: MOSI, pin 6
// - CLK: ATmega164: SCK, pin 8
// - Note use of tri-state buffers to share single SPI channel
//   to program ATmega164 and control LED strip.
//
//Variables:
// - LED_brightness - set intensity from 0 to 127
// - segment_delay  - delay between LED RGB segments
// - strip_delay    - delay between LED strip update
//
//Notes:
// - SPI must be configured for Most significant bit (MSB) first
// - LED brightness is seven bits.  Most significant bit (MSB)
//   must be set to logic one
// - Each LED requires a separate R-G-B intensity components.  The order
//   of data is G-R-B.
// - After sending data for all strip LEDs.  A byte of (0x00) must
//   be sent to return strip to first LED.
// - Data stream for each LED is:
//1-G6-G5-G4-G3-G2-G1-G0-1-R6-R5-R4-R3-R2-R1-R0-1-B6-B5-B4-B3-B2-B1-B0
//
//This example code is in the public domain.
//********************************************************************
```

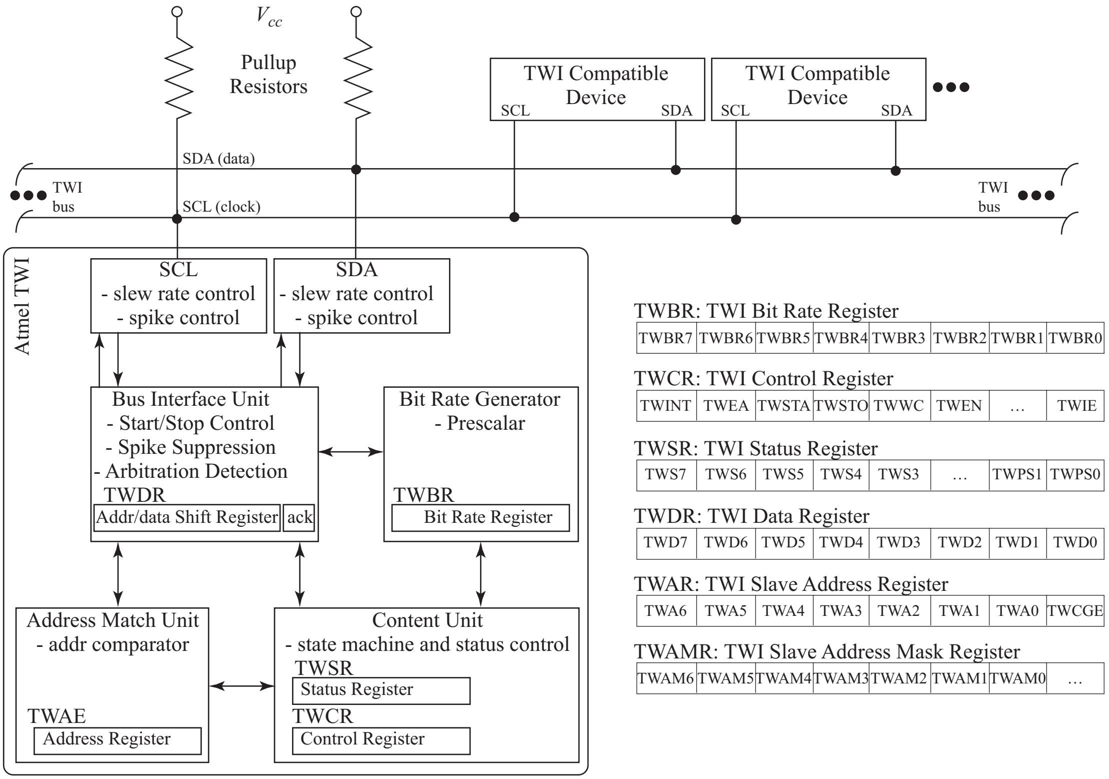
```
#define LED_strip_latch 0x00

//function prototypes
void spi_init(void);
void spi_write(unsigned char byte);
void delay_1s(void);
void delay_100ms(void);
void clear_strip(void);

//Include Files: choose the appropriate include file depending on
//the compiler in use - comment out the include file not in use.

//include file(s) for JumpStart C for AVR Compiler****************
#include<iom164pv.h>                //contains reg definitions

//include file(s) for the Atmel Studio gcc compiler
//#include <avr/io.h>               //contains reg definitions

unsigned char   strip_length = 32; //number of RGB LEDs in strip
unsigned char   LED_brightness;    //0 to 127
unsigned char   position;          //LED position in strip

int main(void)
{
spi_init();
spi_write(LED_strip_latch);       //reset to first segment
clear_strip();                    //all strip LEDs to black
delay_100ms();

//increment the green intensity of the strip LEDs
for(LED_brightness = 0; LED_brightness <= 60;
    LED_brightness = LED_brightness + 10)
   {
   for(position = 0; position<strip_length; position = position+1)
     {
     spi_write(0x80 | LED_brightness);  //Green - MSB 1
     spi_write(0x80 | 0x00);            //Red   - none
     spi_write(0x80 | 0x00);            //Blue  - none
     delay_100ms();
     }
   spi_write(LED_strip_latch);          //reset to first segment
   delay_100ms();
   }

clear_strip();                          //all strip LEDs to black
delay_100ms();

//increment the red intensity of the strip LEDs
for(LED_brightness = 0; LED_brightness <= 60;
    LED_brightness = LED_brightness + 10)
  {
  for(position = 0; position<strip_length; position = position+1)
    {
    spi_write(0x80 | 0x00);            //Green - none
    spi_write(0x80 | LED_brightness);  //Red   - MSB1
    spi_write(0x80 | 0x00);            //Blue  - none

    delay_100ms();
    }
  spi_write(LED_strip_latch);          //reset to first segment
  delay_100ms();
  }

clear_strip();                         //all strip LEDs to black
delay_100ms();

//increment the blue intensity of the strip LEDs
for(LED_brightness = 0; LED_brightness <= 60;
    LED_brightness = LED_brightness + 10)
    {
    for(position = 0; position<strip_length; position = position+1)
      {
      spi_write(0x80 | 0x00);            //Green - none
      spi_write(0x80 | 0x00);            //Red   - none
      spi_write(0x80 | LED_brightness);  //Blue  - MSB1

      delay_100ms();
      }
    spi_write(LED_strip_latch);         //reset to first segment
    delay_100ms();
    }

clear_strip();                         //all strip LEDs to black
delay_100ms();

}

//****************************************************************

void clear_strip(void)
{
//clear strip
for(position = 0; position<strip_length; position = position+1)
  {
  spi_write(0x80 | 0x00);            //Green - none
  spi_write(0x80 | 0x00);            //Red - none
  spi_write(0x80 | 0x00);            //Blue  - none

  spi_write(LED_strip_latch);        //Latch with zero
  delay_100ms();                     //clear delay
  }
}

//*************************************************************
//spi_init: initializes spi system
//*************************************************************
void spi_init()
{
DDRB = 0xB0;            //Set SCK (PB7), MOSI (PB5), PB) for output,
                        //others to input
                        //Configure SPI Control Register (SPCR)
SPCR = 0x5F;            //SPIE:0
                        //SPE: 1  SPI on
                        //DORD:0  MSB first
                        //MSTR:1  Master (provides clock)
                        //CPOL:1  Required by LED strip
                        //CPHA:1  Required by LED strip
                        //SPR1:1  SPR[1:0] 11: div clock 128
                        //SPR0:1
}

//***************************************************************
//spi_write: Used by SPI master to transmit a data byte
//***************************************************************
void spi_write(unsigned char byte)
{
SPDR = byte;
while (!(SPSR & 0x80))
  {
  ;
  }
}

//*****************************************************************
//delay_100ms: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 20 MHz
//  - 100 ms delay requires 2M clock cycles
//  - nop requires 1 clock cycle to execute
//*****************************************************************
void delay_100ms(void)
{
unsigned int i,j;

for(i=0; i < 2000; i++)
  {
  for(j=0; j < 1000; j++)
    {
    asm("nop");
    }
  }
}

//*****************************************************************
//delay_1s: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 20 MHz
//  - 100 ms delay requires 2M clock cycles
//  - nop requires 1 clock cycle to execute
//  - call 10 times for 1s delay
//****************************************************************

void delay_1s(void)
{
unsigned int i;

for(i=0; i< 10; i++)
  {
  delay_100ms();
  }
}

//****************************************************************
```

3.4 TWO-WIRE SERIAL INTERFACE

The TWI subsystem allows the system designer to connect a number of related devices (microcontrollers, transducers, displays, memory storage, etc.) together into a system using a two-wire interconnecting scheme. The TWI allows a maximum of 128 devices to be connected together. Each device has its own unique address and may both transmit and receive over the two-wire bus at frequencies up to 400 kHz. This allows the device to freely exchange information with other devices in a small area network. The TWI is alternately known as the inter-integrated circuit (I^2C) protocol (Philips).

An overview of the TWI system is shown in Figure 3.11. Devices within the small area network are connected by two wires to share data (SDA) and a common clock (SCL). Pullup resistors are required on each of the lines. TWI compatible devices are connected to the SCL and SDA lines as shown.

The microchip TWI system is a state machine to control the “hand shaking” protocol between the TWI master(s) and the multiple slave devices on the TWI bus. If the system contains more than one master designated device, arbitration detection and resolution protocols prevent bus contention. Each slave device has a unique seven bit address to allow one-to-one communication using the Address Match Unit and address comparator. The TWI bus frequency should not exceed 400 kHz. The bus frequency is derived from the microchip microcontroller clock signal using the Bit Rate Generator which contains prescalar hardware. The microchip TWI system also includes signal conditioning features for the SCL and SDA pins including slew rate and spike control.



TWBR: TWI Bit Rate Register

TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0

TWCR: TWI Control Register

TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	…	TWIE

TWSR: TWI Status Register

TWS7	TWS6	TWS5	TWS4	TWS3	…	TWPS1	TWPS0

TWDR: TWI Data Register

TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0

TWAR: TWI Slave Address Register

TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWCGE

TWAMR: TWI Slave Address Mask Register

TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	…

Figure 3.11: TWI system overview [Microchip].

The TWI system is configured and controlled using a series of registers shown in Figure 3.11. Details on specific bit settings are provided in the Microchip ATmega164 datasheet and will not be duplicated here (`www.microchip.com`). These include:

- TWBR: TWI Bit Rate Register
- TWCR: TWI Control Register
- TWSR: TWI Status Register
- TWDR: TWI Data Register
- TWAR: TWI Address Register
- TWAMR: TWI Slave Address Mask Register

Data is exchanged by devices on the TWI bus using s carefully orchestrated "hand shaking" protocol as shown in Figure 3.12. On the left-hand side of the figure are the actions required

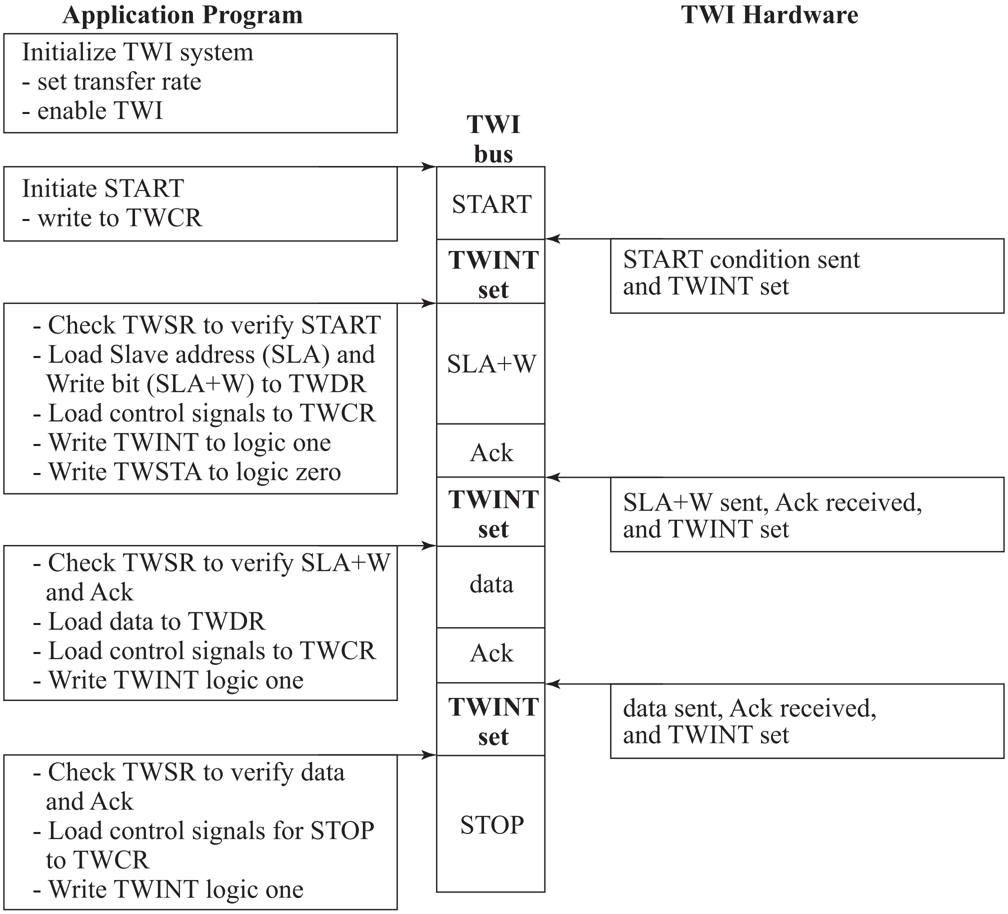
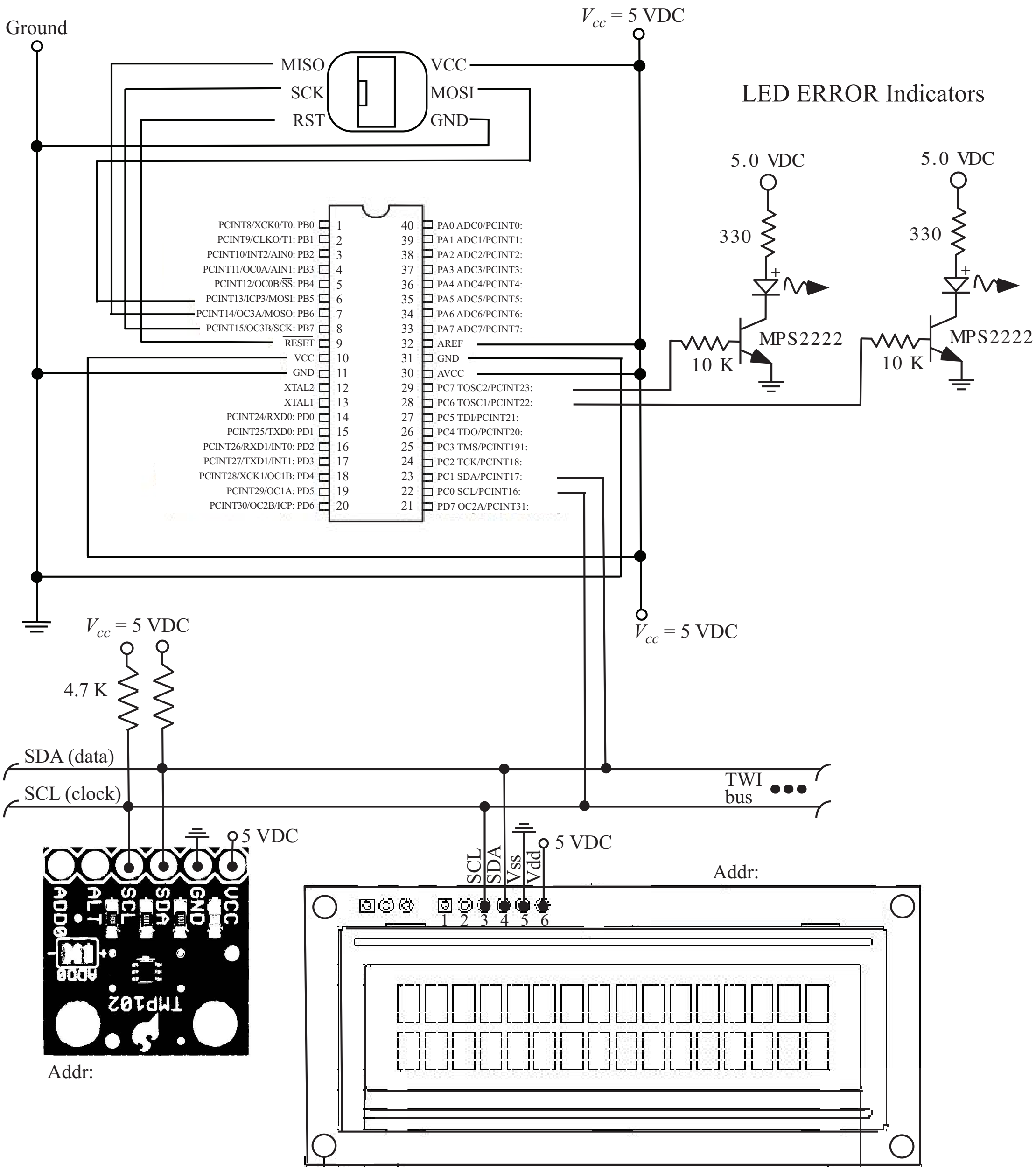

Application Program
TWI Hardware
Initialize TWI system
- set transfer rate
- enable TWI
TWI bus
Initiate START
- write to TWCR
START
TWINT set
START condition sent and TWINT set
- Check TWSR to verify START
- Load Slave address (SLA) and Write bit (SLA+W) to TWDR
- Load control signals to TWCR
- Write TWINT to logic one
- Write TWSTA to logic zero
SLA+W
Ack
TWINT set
SLA+W sent, Ack received, and TWINT set
- Check TWSR to verify SLA+W and Ack
- Load data to TWDR
- Load control signals to TWCR
- Write TWINT logic one
data
Ack
TWINT set
data sent, Ack received, and TWINT set
- Check TWSR to verify data and Ack
- Load control signals for STOP to TWCR
- Write TWINT logic one
STOP

Figure 3.12: TWI operations (`www.microchip.com`).

by the TWI application program and the right side of the figure contains the response from the TWI compatible slave hardware. At each step in the exchange protocol action is initiated by the application program hosted on the TWI master device with a corresponding response from the slave configured device. If the expected response is not received, an error is triggered.

3.4.1 EXAMPLE: TWI COMPATIBLE LCD

In this example a TWI compatible LCD is used to display temperature data from an I2C compatible temperature sensor (TMP102) as shown in Figure 3.13. Note how all devices are connected to the TWI bus via the SDA and SCL pins. Also note the unique TWI address of the LCD and the temperature sensor. In the example we communicate with the LCD (TWI write).



Newhaven NHD-0216K3Z-FL-GBW-V3
Note: for TWI operation short R1 jumper and open R2 jumper

Figure 3.13: TWI connecting sensor to LCD (www.sparkfun.com).

We defer discussion of the TMP102 (TWI read) operation to Chapter 4, the analog-to-digital conversion chapter.

```
//***************************************************************
//twi.c
//
//Example provides twi communication with
//twi configured LCD.
// - LCD: Newhaven NHD-0216K3Z-FL-GBW-V3
// - LCD short R1 jumper, open R2 jumper
//Adapted from Microchip provided TWI
//examples [www.microchip.com]
//***************************************************************

//Include Files: choose the appropriate include file depending on
//the compiler in use - comment out the include file not in use.

//include file(s) for JumpStart C for AVR Compiler****************
//#include<iom164pv.h>           //contains reg definitions

//include file(s) for the Atmel Studio gcc compiler
#include <avr/io.h>              //contains reg definitions

//TWSR status codes with prescaler = 0
#define START              0x08  //START condition transmitted
#define START_REP          0x10  //Repeated START transmitted
#define MT_SLA_NO_ACK      0x20  //SLA+W has been transmitted,
                                 //NOT ACK has been received
#define MT_SLA_ACK         0x18  //SLA+W has been transmitted,
                                 //ACK has been received
#define MT_DATA_ACK        0x28  //Data byte has been transmitted,
                                 //ACK has been received
#define MT_DATA_NO_ACK     0x30  //Data byte has been transmitted,
                                 //NOT ACK has been received
#define MR_SLA_ACK         0x40  //SLA+R has been transmitted,
                                 //ACK has been received
#define ARB_LOST           0x38  //Arbitration lost in SLA+W or
                                 //data bytes

//clock specifications
```

```
#define ceramic_res_freq  20000000UL  //ATmega164 operating freq
#define scl_clock         100000L     //desired TWI bus freq

//peripheral device addresses
#define LCD_twi_addr      0x50 //addr - LSB 0 for write
#define tmp102_twi_addr   0x93
#define TMP_RD            0x93
#define TMP_WR            0x92 //Assume ADR0 is tied to VCC
#define TEMP_REG          0x00


//function prototypes
void initialize_ports(void);
void ERROR(unsigned char error_number);
void LCD_init(void);
void lcd_print_string(char str[]);
void move_LCD_cursor(unsigned char position);
void twi_initialize(void);
void twi_send_byte(unsigned char slave_device_addr,
                   unsigned char send_data);

int main(void)
{
unsigned int i;

initialize_ports();
twi_initialize();
LCD_init();

for(i=0; i<10; i++)
  {
  twi_send_byte(LCD_twi_addr, 'G');
  }

//move cursor to line 2, position 0
move_LCD_cursor(0x40);

lcd_print_string("Test 1 - 2");
}
```

```
//***************************************************************
//function definitions
//***************************************************************

//***************************************************************
//initialize_ports: provides initial configuration for I/O ports
//***************************************************************
void initialize_ports(void)
{
DDRA=0xff;                         //set PORTA[7:0] as output
PORTA=0x00;                        //initialize low

DDRB=0xff;                         //PORTB[7:0] as output
PORTB=0x00;                        //initialize low

DDRC=0xff;                         //set PORTC as output
PORTC=0x00;                        //initialize low

DDRD=0xff;                         //set PORTD as output
PORTD=0x00;                        //initialize low

}

//***************************************************************
//void ERROR - indicates source of error with
//             corresponding LED pattern
//***************************************************************

void ERROR(unsigned char error_number)
{
//Turn off error LEDs
PORTC = 0x00;

if(error_number == 1)        //Error 01
  PORTC = 0x40;
else if(error_number == 2)   //Error 10
  PORTC = 0x80;
else if(error_number == 3)   //Error 11
```

```
  PORTC = 0xC0;
else
  PORTC = 0x00;
}

//*************************************************************
//LCD_init: initializes the USART system
//*************************************************************
void LCD_init(void)
{
twi_send_byte(LCD_twi_addr, 0xFE);
twi_send_byte(LCD_twi_addr, 0x41);              //LCD on

twi_send_byte(LCD_twi_addr, 0xFE);
twi_send_byte(LCD_twi_addr, 0x46);              //cursor to home

twi_send_byte(LCD_twi_addr, 0xFE);
twi_send_byte(LCD_twi_addr, 0x52);
twi_send_byte(LCD_twi_addr,   25);              //set contrast


twi_send_byte(LCD_twi_addr, 0xFE);
twi_send_byte(LCD_twi_addr, 0x53);
twi_send_byte(LCD_twi_addr,    4);              //set backlight
}

//*************************************************************
//void lcd_print_string(char str[])
//*************************************************************

void lcd_print_string(char str[])
{
int k = 0;

while(str[k] != 0x00)
  {
  twi_send_byte(LCD_twi_addr, str[k]);
  k = k+1;
  }
```

```
}

//*************************************************************
//void move_LCD_cursor(unsigned char position)
//*************************************************************

void move_LCD_cursor(unsigned char position)
{
twi_send_byte(LCD_twi_addr, 0xFE);
twi_send_byte(LCD_twi_addr, 0x45);
twi_send_byte(LCD_twi_addr, position);
}

//*************************************************************
//void twi_initialize(void)
//*************************************************************

void twi_initialize(void)
{
//set twi frequency to 100 kHz with ceramic
//resonator frequency at 20 Mhz.
//twi pre-scalar = 1

TWSR = 0;              //no pre-scale
TWBR = ((ceramic_res_freq/scl_clock)-16)/2;
TWCR = TWCR | 0x04;    //TWEN = 1
}

//*************************************************************
//void twi_send_byte(unsigned char slave_device_addr,
//                   unsigned char send_data);
//*************************************************************

void twi_send_byte(unsigned char slave_device_addr,
                   unsigned char send_data)
{
//Send START condition
TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
```

```
//Wait for TWINT Flag set. This indicates that the START
//condition has been transmitted
while (!(TWCR & (1<<TWINT)));

//Check value of TWI Status Register. Mask prescaler bits.
//If status different from START go to ERROR
if ((TWSR & 0xF8) != START)
   ERROR(1);

//Load SLA_W into TWDR Register. Clear TWINT bit in
//TWCR to start transmission of address
TWDR = slave_device_addr;
TWCR = (1<<TWINT) | (1<<TWEN);

//Wait for TWINT Flag set. This indicates that the SLA+W has
//been transmitted, and ACK/NACK has been received.
while (!(TWCR & (1<<TWINT)));

//Check value of TWI Status Register. Mask prescaler bits.
//If status different from MT_SLA_ACK go to ERROR
if((TWSR & 0xF8) != MT_SLA_ACK)
   ERROR(2);

//Load DATA into TWDR Register. Clear TWINT bit in TWCR
//to start transmission of data
TWDR = send_data;
TWCR = (1<<TWINT) | (1<<TWEN);

//Wait for TWINT Flag set. This indicates that the
//DATA has been transmitted, and ACK/NACK has been received.
while (!(TWCR & (1<<TWINT)));

//Check value of TWI Status Register. Mask prescaler bits.
//If status different from MT_DATA_ACK go to ERROR
if((TWSR & 0xF8) != MT_DATA_ACK)
   ERROR(3);

//Transmit STOP condition
TWCR = (1<<TWINT)|(1<<TWEN) | (1<<TWSTO);
```

```
}

//*************************************************************
```

3.5 LABORATORY EXERCISE

Develop a summary table of features for the USART, SPI, and TWI serial communication systems. Develop a two way communication system between two ATmega164s. Defend your choice of system.

3.6 SUMMARY

In this chapter, we have discussed the differences between parallel and serial communications and key serial communication-related terminology. We then, in turn, discussed the operation of the USART, SPI, and TWI serial communication systems. We also provided basic code examples to communicate with the USART, SPI, and TWI systems.

3.7 REFERENCES AND FURTHER READING

Atmel 8-bit AVR Microcontroller with 16/32/64/128K Bytes In-System Programmable Flash, ATmega164A, ATmega164PA, ATmega324A, ATmega324PA, ATmega644A, ATmega644PA, ATmega1284, ATmega1284P, 8272C-AVR-06/11, data sheet: 8272C-AVR-01/15, Atmel, San Jose, CA, 2015. 54, 87

Barrett, S. F. and Pack, D. J. *Microcontroller Fundamentals for Engineers and Scientists*. Morgan & Claypool, San Rafael, CA, 2006. DOI: 10.2200/s00025ed1v01y200605dcs001. 53

Garmin International, Inc., 2019. www.garmin.com 75

3.8 CHAPTER PROBLEMS

1. Summarize the differences between parallel and serial communications.

2. Summarize the differences among the USART, SPI, and TWI methods of serial communication.

3. Draw a block diagram of the USART system, label all key registers, and all keys USART flags.

4. Draw a block diagram of the SPI system, label all key registers, and all keys USART flags.

5. If an ATmega164 microcontroller is operating at 12 MHz, what is the maximum transmission rate for the USART and the SPI?

6. What is the ASCII encoded value for “Claypool”? With odd parity? Even parity?

7. Draw the schematic of a system consisting of two ATmega164 that will exchange data via the SPI system. The system should include RS-232 level shifting.

8. Write the code to implement the system described in question seven.

9. Add USART, SPI, and TWI features to the testbench software introduced in Chapter 1.

CHAPTER 4

Analog-to-Digital Conversion

Objectives: After reading this chapter, the reader should be able to:

- explain the difference between analog and digital signals;
- illustrate the analog-to-digital converter (ADC) conversion process;
- assess the quality of an ADC using the metrics of sampling rate, quantization levels, number of bits used for encoding, and dynamic range;
- design signal conditioning circuits to interface sensors with ADCs;
- describe the key registers used in the ATmega164 ADC;
- describe the steps to perform by an ADC on the ATmega164;
- program the ATmega164 ADC;
- describe the digital-to-analog converter (DAC); and
- equip the ATmega164 with the digital-to-analog conversion capability.

4.1 OVERVIEW

A microcontroller is used to process information obtained from the natural world, decide on a course of action based on the information collected, and then issue control signals to implement the decision. Because much of the information from the natural world is analog or continuous in nature and the microcontroller is a digital or discrete-based processor, a method to convert an analog signal to a corresponding digital one is required [Barrett and Pack, 2008]. An ADC system performs this task, whereas a DAC performs the conversion in the opposite direction. We will discuss both types of converters in this chapter.

In the first section, the fundamental concepts associated with the ADC process are presented. The conversion process, followed by a presentation of different hardware implementations of the process are covered next. Much of these early sections contain the same material the reader would find in the text, *Microcontroller Fundamentals for Engineers and Scientists*. The basic features of the ATmega164 ADC system are then reviewed, followed by a system description and a discussion of key ADC registers. The chapter concludes with several illustrative ADC code examples and a discussion of the DAC process.

4.2 BACKGROUND THEORY

Before discussing the ADC process, some definitions on analog and digital signals are in order.

4.2.1 ANALOG VS. DIGITAL SIGNALS

A signal is a collection of values representing the state of a physical variable. The collection can be as small as a single value or can have an infinite number of values. Values are usually arranged in order, e.g., over time or over a spatial axis, to display the information. The time and spatial variables are called independent variables, because they are not affected by the physical variables of interest. For example, temperature change may be monitored over time. The temperature measured is dependent on the time, not the other way around. Figure 4.1 shows an altitude trajectory of a bird flying over a short period. The signal shows how the altitude of the bird changes continuously.

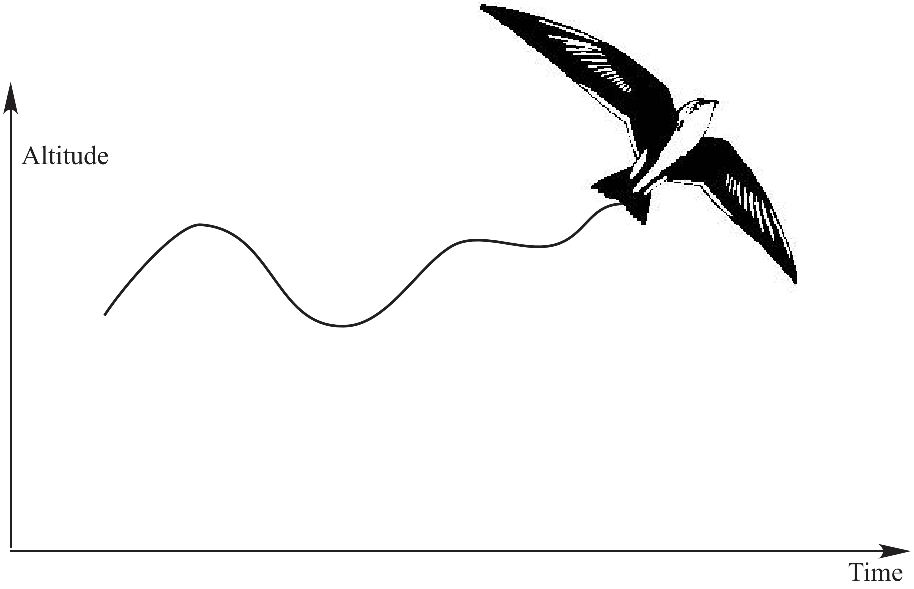


Figure 4.1: Altitude trajectory generated by a flying bird.

Figure 4.2 shows a grayscale image of a six-legged robot. The image captured the light intensities of the scene using a charge-coupled device camera. As we move from the left to the right on the image and observe the intensity changes, we can find vertical edges by detecting signal intensity changes in the spatial axis. The same analysis can be performed as we move spatially from the top to the bottom of the image.

Analog signals are those whose physical variable values change continuously over their independent variable. Most physical variables, your speech characteristics, movement of stars,

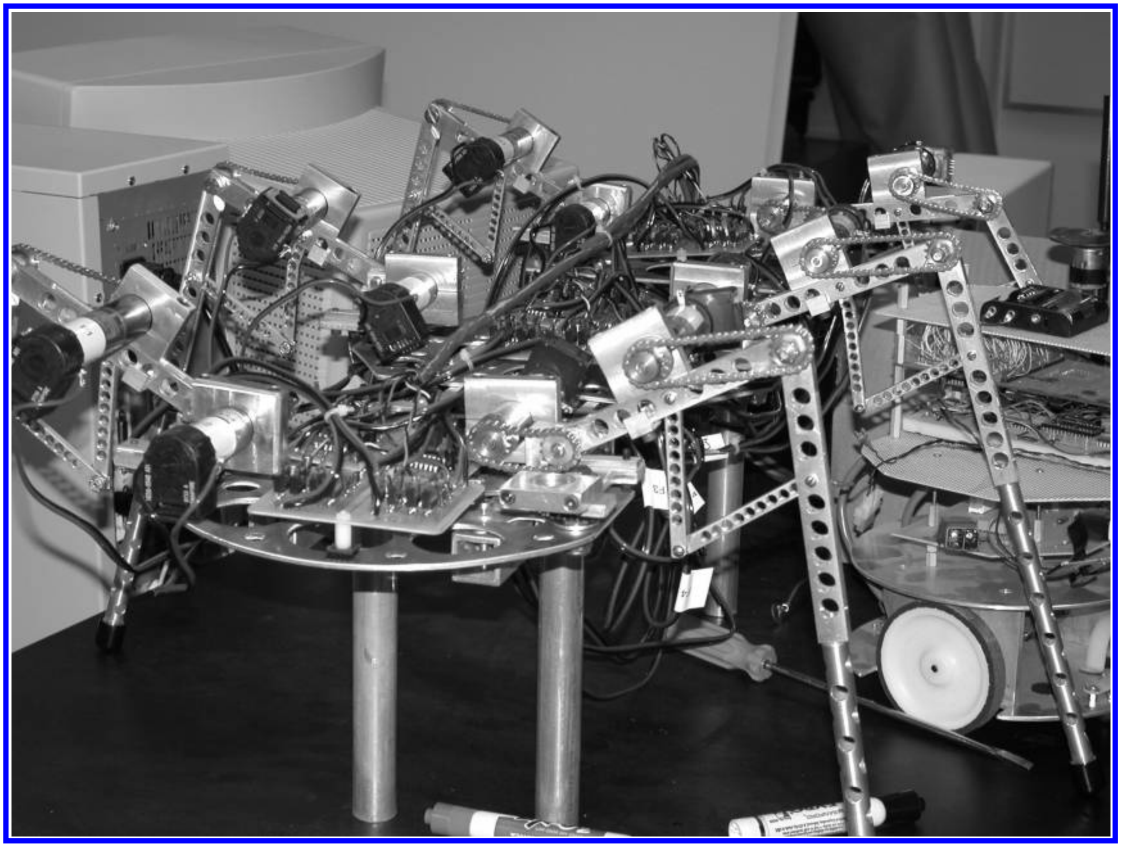


Figure 4.2: A photo of a walking robot.

and the music you hear at a concert are all examples of analog signals. Digital signals, on the other hand, have their physical variables defined only for discrete instances over their independent variables. Although it may look continuous to human eyes, the photo example shown in Figure 4.2 is a discrete signal because pixels (picture elements) that make up a camera image cannot capture all space within the camera frame. The image is only a finite composition of discrete intensity values seen by a discrete number of pixels.

Digital signals are important because all signals represented in digital systems, computers, and microcontrollers are in digital forms. The important task is how to faithfully represent analog signals using digital signals. For example, human voices must be converted to corresponding digital signals before they can be routed by digital switching circuits in telephone communication systems. Similarly, voice commands to robots must be converted to a digital form before robots can process the command.

As shown in the previous examples, we live in an analog world; that is, physical variables are analog signals. It is precisely this reason why the ADC is so very important in any digital systems that interact with an analog environment.

4.2.2 SAMPLING, QUANTIZATION, AND ENCODING

In this subsection, three important processes associated with the ADC are presented, starting with the subject of sampling. Imagine yourself as a photographer in an Olympic diving stadium. Your job is taking a sequence of pictures of divers jumping from a diving board 10 m above the surface of the diving pool. Your goal is to put the sequence of pictures together to reconstruct the motion of each diver. The sequence of pictures makes up samples of a divers' motions. If a diver tries a complex dive and you want to faithfully reconstruct the motion, you must take enough pictures from the start to the end of the dive. If a diver makes a simple routine dive, you only need to take a few pictures over the period of the dive. Two very different cases of motions generated by a diver are shown in Figure 4.3. The same time sequence is used to capture samples for both dives. As can be seen from the figure, the frame (a) motion cannot be regenerated from the samples, whereas the dive shown in frame (b) can clearly be reconstructed from the same number of samples used to capture both motions.

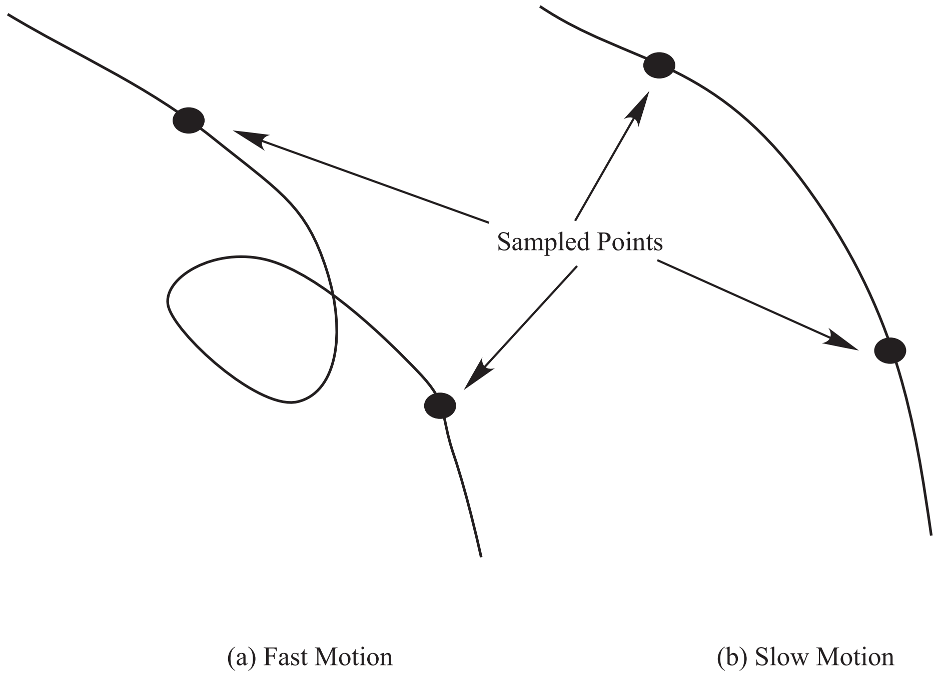


Figure 4.3: Two divers jumping off the platforms: (a) fast motion and (b) slow motion.

Sampling is the process of taking "snapshots" of a signal over time. Naturally, when we sample a signal, we want to sample it in an optimal fashion such that we can capture the essence of the signal while minimizing the use of resources. In essence, we want to minimize the number of samples while faithfully reconstructing the original signal from the samples. As can be deduced from our diving example above, the rate of change in a signal determines the number of samples required to faithfully reconstruct the signal, provided that all adjacent samples are captured with the same sample timing intervals.

Harry Nyquist from Bell Laboratory studied the sampling process and derived a criterion that determines the minimum sampling rate needed for any continuous analog signal. His, now famous, minimum sampling rate is known as the Nyquist sampling rate, which states that one must sample a signal at least twice as fast as the highest frequency content of the signal of interest. For example, if we are dealing with the human voice signal that contains frequency components that span from about 20 Hz to 4 kHz, the Nyquist sample theorem requires that we must sample the signal at least at 8 kHz, 8000 "snapshots" every second. For further study on the Nyquist sampling rate, please refer to Barrett and Pack [2006]. Sampling is important because when we want to represent an analog signal in a digital system, such as a computer, we must use the appropriate sampling rate to capture the analog signal for a faithful representation in digital systems.

Now that we understand the sampling process, let us move on to the second process of the ADC; quantization. Each digital system has a number of bits, which it uses as the basic units, to represent data. A bit is the most basic unit where single binary information, 1 or 0, is represented. A nibble is made up of 4 bits put together. A byte is 8 bits and 2 nibbles make a byte!

In the previous section, we avoided a detailed discussion of the form of captured signal samples. When an analog signal is sampled, digital systems need some means to represent the captured samples. The quantization of a sampled signal is how the signal is represented as one of "n" quantization levels. For example, suppose you have a single bit to represent an incoming signal. You only have two different numbers, 0 and 1, available to represent the digitized signal. You may say that you can distinguish only logic low (0) from high (1). Suppose you have two bits. You can represent four different levels using the following binary encoding scheme: 00, 01, 10, and 11. What if you have three bits? You now can represent eight different discrete levels: 000, 001, 010, 011, 100, 101, 110, and 111. Think of it as follows. When you had two bits, you were able to represent four different levels. If we add one more bit, that bit can be 1 or 0, making the total possibilities to eight. Similar discussion can lead us to conclude that given n bits, we have 2^n different numbers or discrete levels one can represent.

Figure 4.4 shows how n bits are used to quantize a range of values. In many digital systems, the incoming signals are voltage signals. The voltage signals are first obtained from physical signals with the help of transducers, such as microphones, angle sensors, and infrared sensors. The voltage signals are then conditioned to map their range to the input range of a digital system,

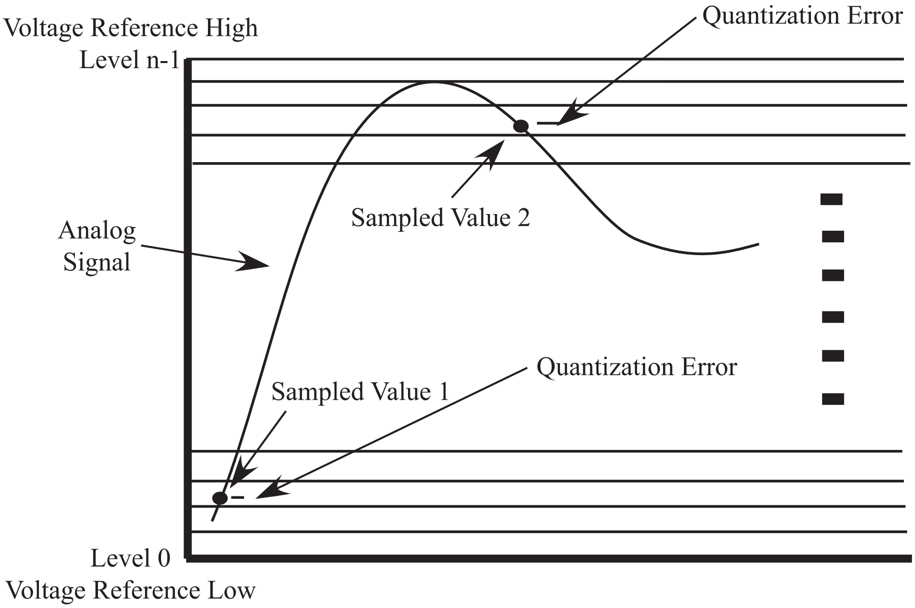


Figure 4.4: Quantization.

typically 0–5 V. In Figure 4.4, n bits allow you to divide the input signal range of a digital system into 2^n different quantization levels. As can be seen from the figure, higher quantization levels mean better mapping of an incoming signal to its true value. If we only had a single bit, we can only represent levels 0 and 1. Any analog signal value in between the range has to be mapped either as level 0 or level 1. This does not provide many choices.

Now imagine what happens as we increase the number of bits available for the quantization levels. What happens when the available number of bits is 8? How many different quantization levels are available now? Yes, 256. How about 10, 12, or 14? Notice also that as the number of bits used for the quantization levels increases for a given input range the "distance" between two adjacent levels decreases with a factor of a polynomial.

Finally, the encoding process involves converting a quantized signal into a digital binary number. Suppose again we are using eight bits to quantize a sampled analog signal. The quantization levels are determined by the 8 bits, and each sampled signal is quantized as one of 256 quantization levels. Consider the two sampled signals shown in Figure 4.5. The first sample is mapped to quantization level 2 and the second one is mapped to quantization level 198. Note the amount of quantization error introduced for both samples. Now consider Figure 4.5. The same signal is sampled at the same time but quantized using a less number of bits. Note that the quantization error is inversely proportional to the number of bits used to quantize the signal.

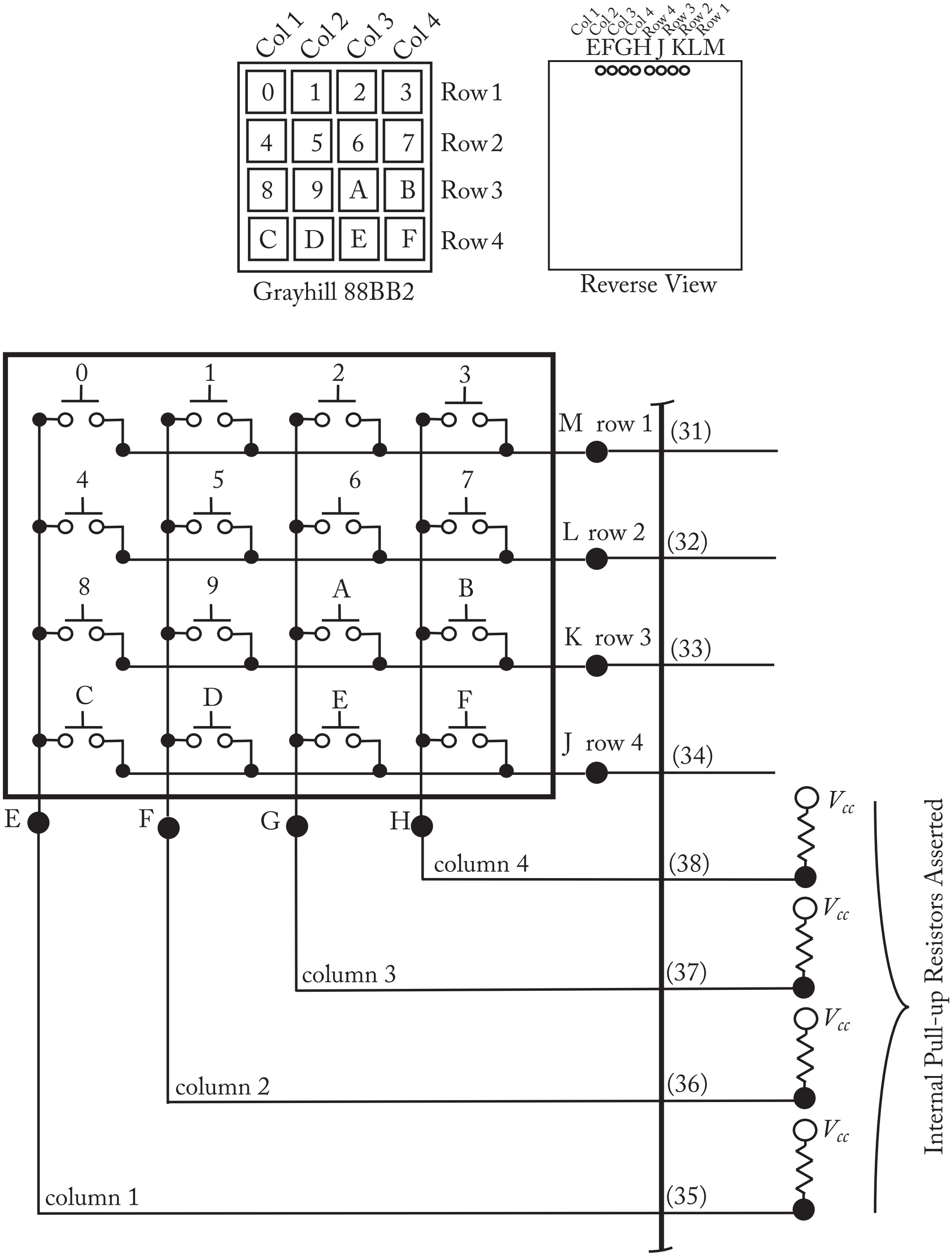


Figure 4.5: Quantization with fewer bits.

Once a sampled signal is quantized, the encoding process involves representing the quantization level with the available bits. Thus, for the first sample, the encoded sampled value is 0000_0001 (or 0x01 in hexadecimal), whereas the encoded sampled value for the second sample is 1100_0110 (or 0xC6 in hexadecimal). As a result of the encoding process, sampled analog signals are now represented as a set of binary numbers. Thus, the encoding is the last necessary step to represent a sampled analog signal into its corresponding digital form, as shown in Figure 4.6.

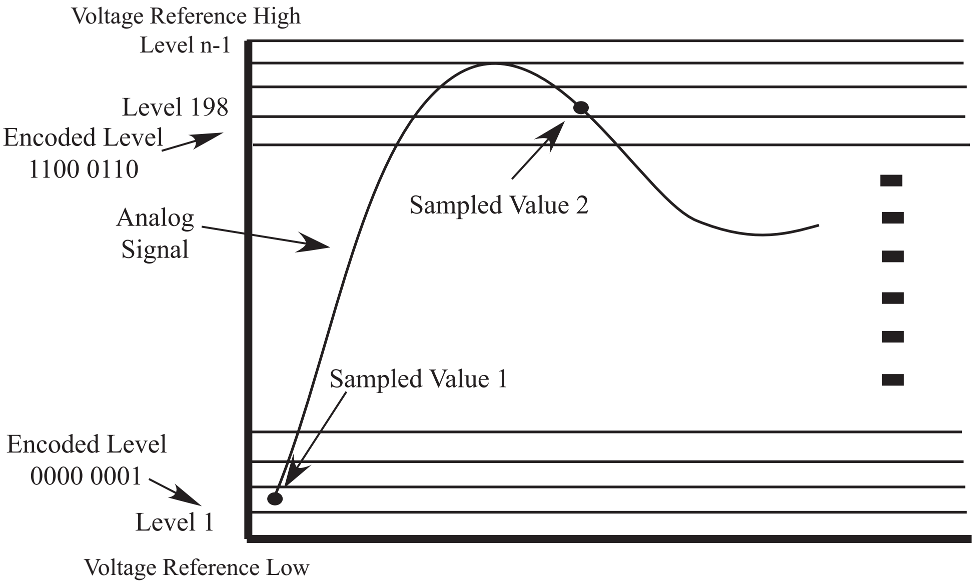


Figure 4.6: Encoding.

4.2.3 RESOLUTION AND DATA RATE

Resolution is a measure used to quantize an analog signal. In fact, resolution is nothing but the "distance" between two adjacent quantization levels we discussed earlier. Suppose again we have a range of 5 V and a single bit to represent an analog signal. The resolution in this case is 2.5 V, a very poor resolution. You can imagine how your television screen will look if you only had only two levels to represent each pixel, black and white. The maximum error, called the resolution error, is 2.5 V for the current case, 50% of the total range of the input signal. Suppose you now have four bits to represent quantization levels. The resolution now becomes 1.25 V, or 25% of the input range. Suppose you have 20 bits for quantization levels. The resolution now becomes 4.77×10^{-6}, 9.54×10^{-5}% of the total range. The discussion we presented simply illustrates that as we increase the available number of quantization levels within a range, the distance between adjacent levels decreases, reducing the quantization error of a sampled signal. As the number of quantization levels increases, the error decreases, making the representation of a sampled analog signal more accurate in the corresponding digital form. The number of bits used for the quantization is directly proportional to the resolution of a system.

Now let us move onto the discussion of the data rate. The definition of the data rate is the amount of data generated by a system per some time unit. Typically, the number of bits or the number of bytes per second is used as the data rate of a system. We just saw that the more bits we use for the quantization levels, the more accurate we can represent a sampled analog signal. Why not use the maximum number of bits current technologies can offer for all digital systems, when we convert analog signals to digital counterparts? It has to do with the cost involved. For example, suppose you are working for a telephone company and your switching system must accommodate 100,000 customers. For each individual phone conversation, suppose the company uses an 8-kHz sampling rate and you are using 10 bits for the quantization levels for each sampled signal.[1] If all customers are making out-of-town calls, what is the number of bits your switching system must process to accommodate all calls? The answer will be 100,000 × 8000 × 10, or 8 billion bits per every second! You will need some major computing power to meet the requirement. For such reasons, when designers make decisions on the number of bits used for the quantization levels, they must consider the computational burden the selection will produce on the digital system vs. the required system resolution.

You will also encounter the term *dynamic range* when you consider finding appropriate ADCs. The dynamic range is a measure used to describe the signal to noise ratio. The unit used for the measurement is decibel, which is the strength of a signal with respect to a reference signal. The greater the decibel number, the stronger the signal is compared with a noise signal. The definition of the dynamic range is $20 \log 2^b$, where b is the number of bits used to convert analog signals to digital signals. Typically, one will find 8 to 12 bits used in commercial ADCs, translating the dynamic range from $20 \log 2^8$ dB to $20 \log 2^{12}$ dB [Oppenheim and Schafer, 1999].

4.3 ANALOG-TO-DIGITAL CONVERSION PROCESS

The goal of the ADC is to accurately represent analog signals as digital signals. Toward this end, three signal processing procedures—sampling, quantization, and encoding—described in the previous section must be combined together. Before the coversion process takes place, we first need to convert a physical signal into an electrical signal with the help of a transducer. A transducer is an electrical and/or mechanical system that converts physical signals into electrical signals or electrical signals to physical signals. Depending on the purpose, we categorize a transducer as an input transducer or an output transducer. If the conversion is from physical to electrical, we call it an input transducer. The mouse, the keyboard, and the microphone for your

[1] For the sake of our discussion, we ignore other overhead involved in processing a phone call such as multiplexing, demultiplexing, and serial-to-parallel conversion.

computer all fall under this category. A camera, an infrared sensor, and a temperature sensor are also input transducers.

The output transducer converts electrical signals to physical signals. The computer screen and the printer for your computer are output transducers. Speakers and electrical motors are also output transducers. Therefore, transducers play the central part for digital systems to operate in our physical world by transforming physical signals to electrical signals and electrical signals to physical ones.

In addition to transducers, we also need signal conditioning circuitry, before we apply the analog-to-digital process or its opposite, the digital-to-analog process. The signal conditioning circuitry is called the transducer interface. The objective of the transducer interface circuit is to scale and shift the electrical signal range to map the output of the input transducer to the input of the ADC. Figure 4.7 shows the transducer interface circuit using an input transducer.

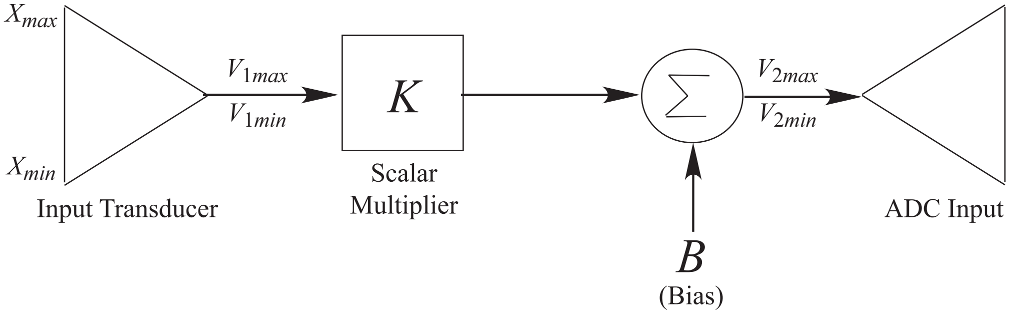


Figure 4.7: A block diagram of the signal conditioning circuit for an ADC. The range of the sensor voltage output is mapped to the ADC input voltage range. The scalar multiplier maps the magnitudes of the two ranges, and the bias voltage is used to align two limits.

The output of the input transducer is first scaled by constant K. In the figure, we use a microphone as the input transducer whose output ranges from −5 to +5 VDC. The input to the ADC ranges from 0–5 VDC. The box with constant K maps the output range of the input transducer to the input range of the converter. Naturally, we need to multiply all input signals by 1/2 to accommodate the mapping. Once the range has been mapped, the signal now needs to be shifted. Note that the scale factor maps the output range of the input transducer as −2.5 to +2.5 VDC instead of 0–5 VDC. The second portion of the circuit shifts the range by 2.5 VDC, thereby completing the correct mapping. Actual implementation of the circuit components is accomplished using operational amplifiers with some feedback loops.

In general, the scaling and bias process may be described by two equations:

$$V_{2\,\mathrm{max}} = (V_{1\,\mathrm{max}} \times K) + B$$
$$V_{2\,\mathrm{min}} = (V_{1\,\mathrm{min}} \times K) + B.$$

The variable $V_{1\,\max}$ represents the maximum output voltage from the input transducer. This voltage occurs when the maximum physical variable ($X_{\max}$) is presented to the input transducer. This voltage must be scaled by the scalar multiplier (K) and then have a DC offset bias voltage (B) added to provide the voltage, $V_{2\,\max}$, to the input of the ADC.

Similarly, variable $V_{1\,\min}$ represents the minimum output voltage from the input transducer. This voltage occurs when the minimum physical variable ($X_{\min}$) is presented to the input transducer. This voltage must be scaled by the scalar multiplier (K) and then have a DC offset bias voltage (B) added to produce voltage $V_{2\,\min}$ to the input of the ADC.

Usually, the values of $V_{1\,\max}$ and $V_{1\,\min}$ are provided with the documentation for the transducer. Also, the values of $V_{2\,\max}$ and $V_{2\,\min}$ are known. They are the high and low reference voltages for the ADC system (typically 5 and 0 VDC for microchip microcontrollers). We thus have two equations and two unknowns to solve for K and B. The circuits to scale by K and add the offset B are usually implemented with operational amplifiers. An implied assumption is the transducer has a linear response relating the measured physical variable to a corresponding electrical response. We refer interested readers to the book by Thomas and Rosa [2003].

Once a physical signal has been converted to its corresponding electrical signal with the help of an input transducer and the output of the transducer mapped correctly to the input of the ADC, the ADC process can start. The first step of the ADC is the sampling of the analog signal. When selecting a converter, one must consider the type of physical signal that is being converted to properly ensure the required sampling rate. As discussed in the previous section, using the proper sampling rate is the first step that determines whether an analog signal will be represented correctly in digital systems. What this means for the reader is to select an ADC that can handle a required conversion rate. Because most microcontrollers now come with a built-in ADC, one must study the user manual portion discussing the conversion rate and make sure that the required sampling rate for the application falls under the advertised conversion rate. For example, if you need to convert the signal representing a person's blood pressure, the sampling rate with 100 Hz (100 samples per second) will suffice. On the other hand, if you are dealing with human voice signals, you need at least an 8-kHz sampling rate capacity (see Enderle et al. [2000] for details).

Once the analog signal has been sampled, the quantization process takes place. For example, if we can only operate with maximum quantization error of 0.01 V, we need to choose a converter with, at minimum, 10 bits (about 5 mV). To determine the number of bits and its corresponding maximum quantization error, we use the following equation:

$$\text{Resolution} = \frac{\text{range}}{2^b}.$$

Once the quantization level has been determined, we can now encode it using the available bits. As seen in the previous section, the process is a simple conversion of a decimal number (quantization level) to a binary number.

In summary, the ADC process has three steps for completion: sampling, quantization, and encoding. In the next section, we delve into four different technologies used to implement the ADC process.

Example: A photodiode is a semiconductor device that provides an output current corresponding to the light impinging on its active surface. The photodiode is used with a transimpedance amplifier to convert the output current to an output voltage. A photodiode/transimpedance amplifier provides an output voltage of 0 V for maximum rated light intensity and –2.50 VDC of output voltage for the minimum rated light intensity. Calculate the required values of K and B for this light transducer, so it may be interfaced to a microcontroller's ADC system:

$$\begin{aligned}
V_{2\,\max} &= (V_{1\,\max} \times K) + B \\
V_{2\,\min} &= (V_{1\,\min} \times K) + B \\
5.0\text{ V} &= (0\text{ V} \times K) + B \\
0\text{ V} &= (-2.50\text{ V} \times K) + B.
\end{aligned}$$

The values of K and B may then be determined to be 2 and 5 VDC, respectively.

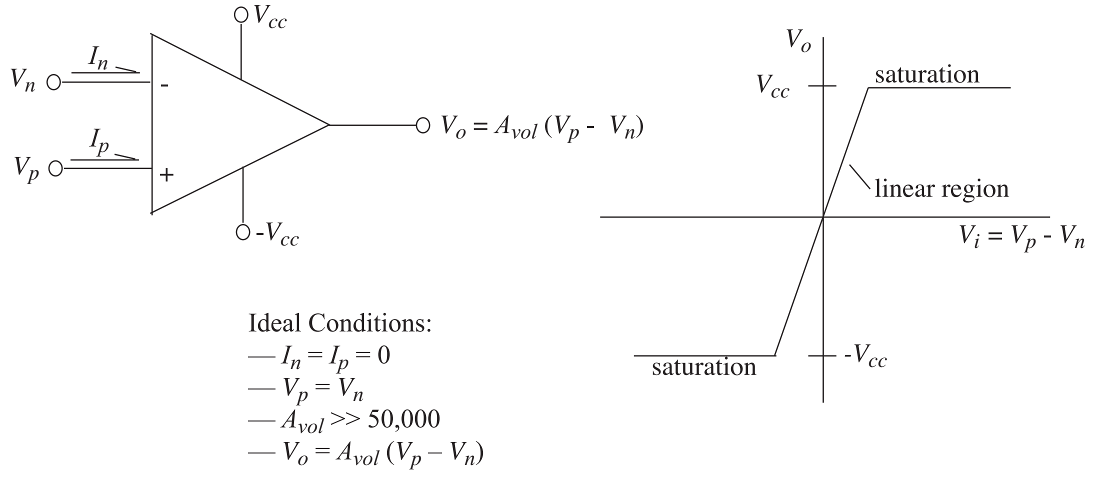
4.3.1 OPERATIONAL AMPLIFIERS

In the previous section we discussed the transducer interface design (TID) process. This process yields a required value of gain (K) and DC bias (B). Operational amplifiers (op amps) are typically used to implement a TID interface. In this section we briefly introduce operational amplifiers including ideal op amp characteristics, classic op amp circuit configurations, and an example to illustrate how to implement a TID with op amps. Op amps are also used in a wide variety of other applications including analog computing, analog filter design, and a myriad of other applications. We do not have the space to investigate all of these related applications. The interested reader is referred to Faulkenberry [1977] for pointers to some excellent texts on this topic.

The Ideal Operational Amplifier

A diagram and operational characteristics of a generic ideal operational amplifier are illustrated in Figure 4.8. An ideal operational does not exist in the real world. However, it is a good first approximation for use in developing op amp application circuits.

The op amp is an active device (requires power supplies) equipped with two inputs, a single output, and several voltage source inputs. The two inputs are labeled Vp, or the non-inverting input, and Vn, the inverting input. The output of the op amp is determined by taking the difference between Vp and Vn and multiplying the difference by the open loop gain (A_{vol}) of the op amp, which is typically a large value much greater than 50,000. Due to the large value of A_{vol}, it does not take much of a difference between Vp and Vn before the op amp will saturate. When an op amp saturates it does not damage the op amp but the output is limited



Ideal Conditions:
— $I_n = I_p = 0$
— $V_p = V_n$
— $A_{vol} >> 50{,}000$
— $V_o = A_{vol}\,(V_p - V_n)$

Figure 4.8: Ideal operational amplifier characteristics.

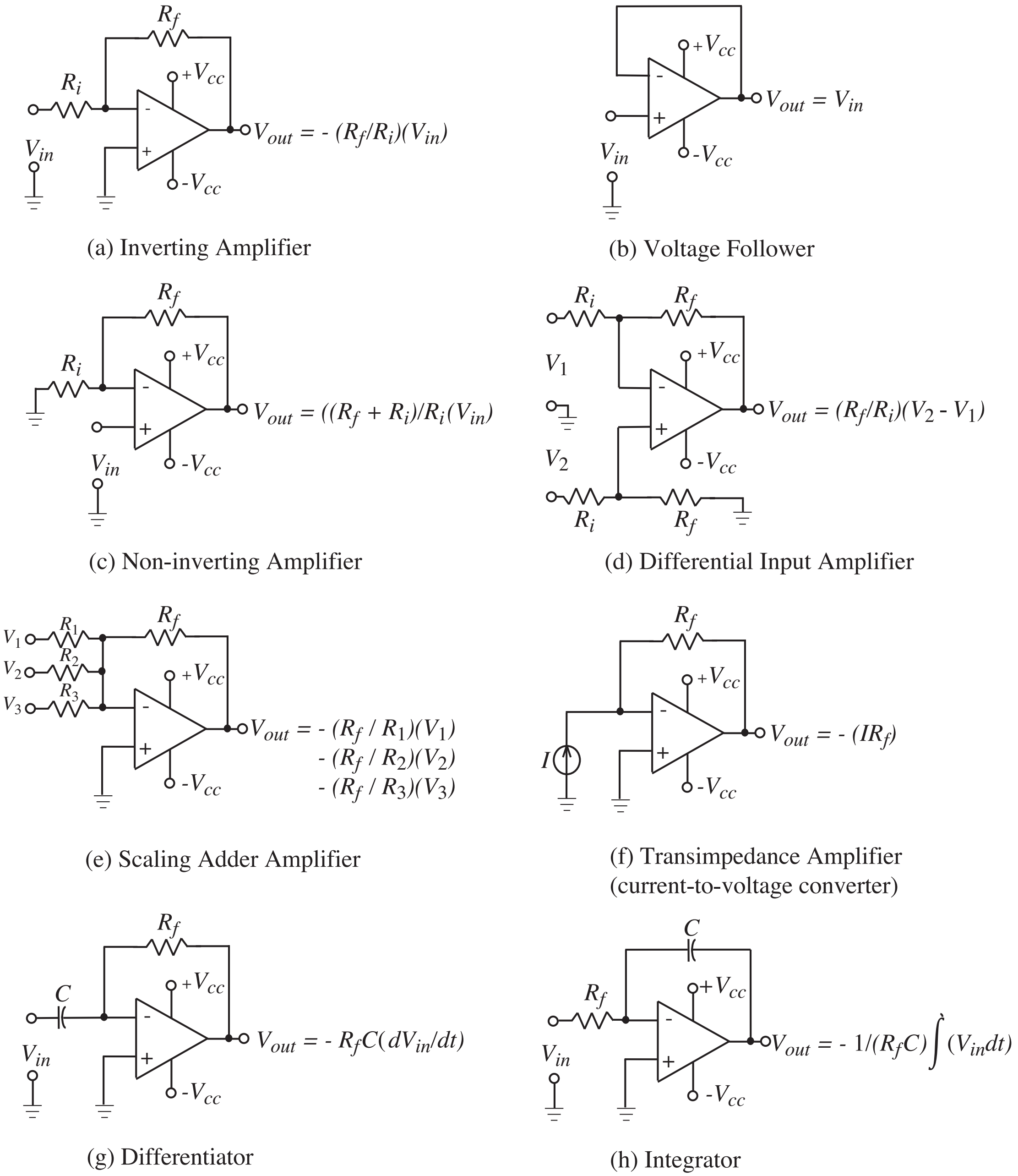
to ±V_{cc}. This will clip the output, and hence distort the signal, at levels approximately one volt short of ±V_{cc}. Op amps are typically used in a closed loop, negative feedback configuration. A sample of classic operational amplifier configurations with negative feedback are provided in Figure 4.9 [Faulkenberry, 1977].

It should be emphasized that the equations provided with each operational amplifier circuit are only valid if the circuit configurations are identical to those shown. Even a slight variation in the circuit configuration may have a dramatic effect on circuit operation. It is important to analyze each operational amplifier circuit using the following steps.

- Write the node equation at node V_n for the circuit.
- Apply ideal op amp characteristics to the node equation.
- Solve the node equation for V_o.

As an example, we provide the analysis of the non-inverting amplifier circuit in Figure 4.10. This same analysis technique may be applied to all of the circuits in Figure 4.9 to arrive at the equations for V_{out} provided.

Example: In the previous section, it was determined that the values of K and B were 2 and 5 VDC, respectively. The two-stage op amp circuitry shown in Figure 4.11 implements these values of K and B. The first stage provides an amplification of -2 due to the use of the non--inverting amplifier configuration. In the next second stage an adding amplifier is used to add the output of the first stage with a bias of -5 VDC. Since this stage also introduces a minus sign to the result, the overall result of a gain of 2 and a bias of $+5$ VDC is achieved.


R_f
R_i
$+V_{cc}$
V_{in}
$V_{out} = - (R_f/R_i)(V_{in})$
$-V_{cc}$
$V_{out} = V_{in}$
$V_{out} = ((R_f + R_i)/R_i(V_{in})$
V_1
V_2
$V_{out} = (R_f/R_i)(V_2 - V_1)$
R_1
R_2
R_3
V_3
$V_{out} = - (R_f / R_1)(V_1)$
$- (R_f / R_2)(V_2)$
$- (R_f / R_3)(V_3)$
I
$V_{out} = - (IR_f)$
C
$V_{out} = - R_fC(dV_{in}/dt)$
$V_{out} = - 1/(R_fC)\int (V_{in}dt)$
-
+

(a) Inverting Amplifier

(b) Voltage Follower

(c) Non-inverting Amplifier

(d) Differential Input Amplifier

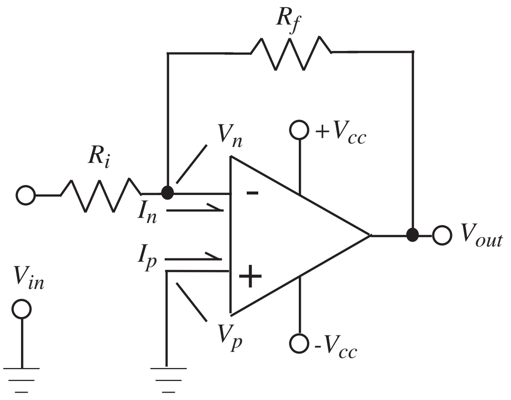
(e) Scaling Adder Amplifier

(f) Transimpedance Amplifier
(current-to-voltage converter)

(g) Differentiator

(h) Integrator

Figure 4.9: Classic operational amplifier configurations. (Adapted from Faulkenberry [1977].)



Node equation at Vn:

$$(V_n - V_{in})R_i + (V_n - V_{out})/R_f + I_n = 0$$

Apply ideal conditions:

$$I_n = I_p + 0$$

$$V_n = V_p = 0 \text{ (since } V_p \text{ is grounded)}$$

Solve node equation for Vout:

$$V_{out} = - (R_f / R_i)(V_{in})$$

Figure 4.10: Operational amplifier analysis for the non-inverting amplifier. (Adapted from Faulkenberry [1977].)

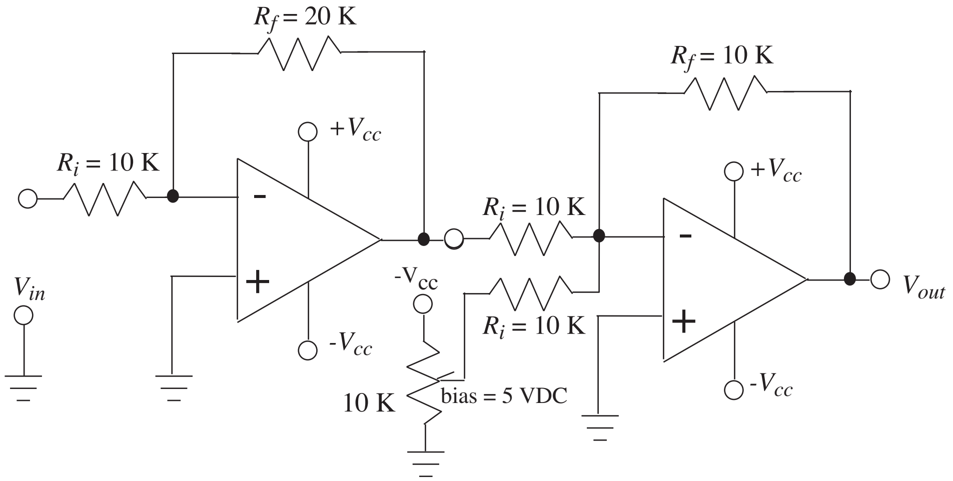


Figure 4.11: Operational amplifier implementation of the transducer interface design (TID) example circuit.

4.4 ADC CONVERSION TECHNOLOGIES

The ATmega164 uses a successive approximation conversion technique to convert an analog sample into a 10-bit digital representation. In this section, we discuss several techniques including the successive approximation method. In certain applications, you are required to use converter technologies external to the microcontroller.

We begin by briefly discussing four different types of technologies used in ADCs: the successive approximation conversion, the integration conversion, the counter-based conversion, and the parallel conversion. For a detailed discussion, we refer the interested readers to Barrett and Pack [2006]. For the following discussion of different ADC technologies, see Figure 4.12. One can group all existing technologies into two different categories: direct conversion vs. indirect conversion. The successive approximation conversion, counter-based conversion, and parallel conversion use techniques to directly convert analog input signals into their digital counterparts. They are all considered to fall under the direct conversion category. The integration method uses conversion time as the means to indirectly measure the magnitude of the incoming analog signals, an indirect approach.

4.4.1 SUCCESSIVE APPROXIMATION

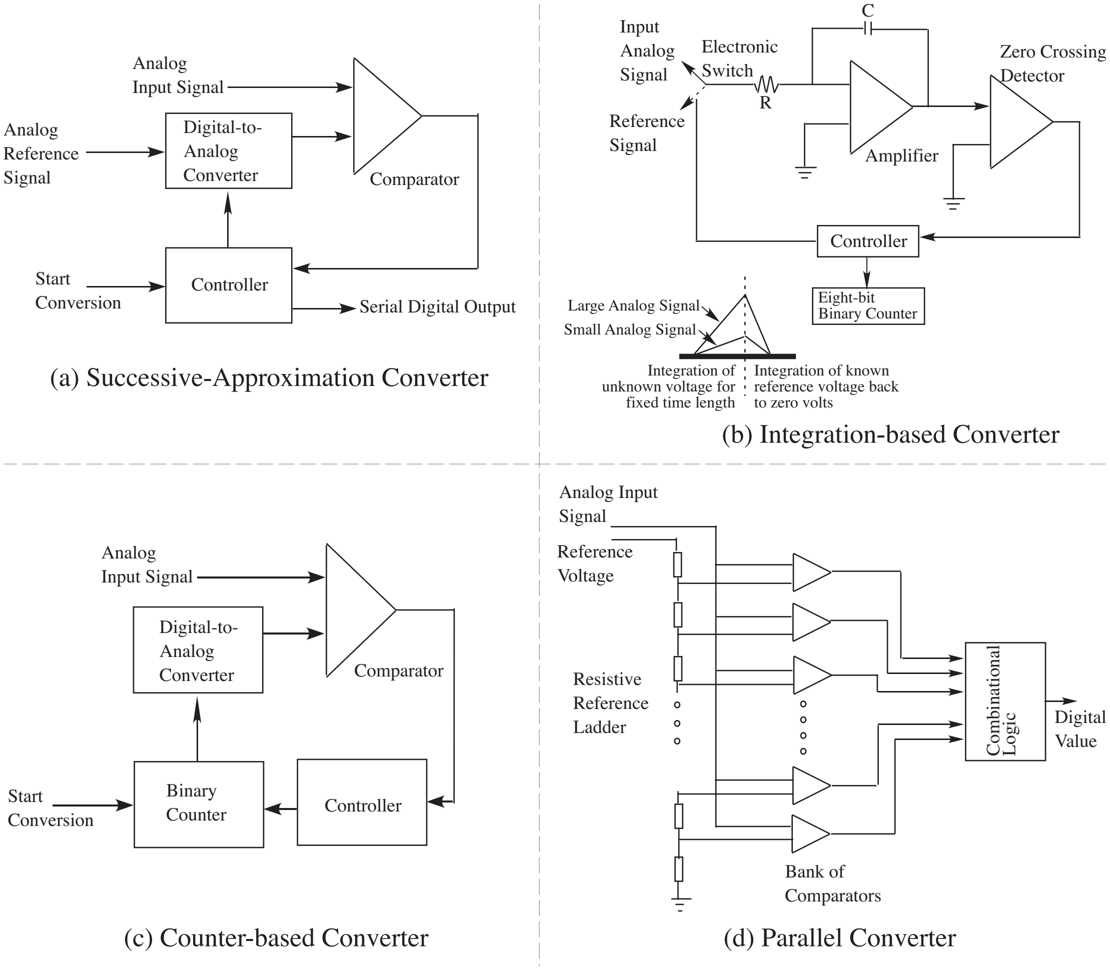
The successive approximation technique uses a DAC, a controller, and a comparator to perform the analog-to-digital process. Starting from the most significant bit (MSB) down to the least significant (LSB), the controller turns on each bit at a time and generates an analog signal, with the help of the DAC, to be compared with the original input analog signal. Based on the result of the comparison, the controller changes or leaves the current bit and turns on the next MSB. The process continues until decisions are made for all available bits.

One can consider the process similar to a game children play. One child thinks of a number in the range of 0–10 and asks another child to guess the number within n turns. The first child will tell the second child whether a guessed number is higher or lower than the answer at the end of each turn. The optimal strategy in such a situation is to guess the middle number in the range, say 5. If the answer is higher than 5, the second guess should be 8. If the answer is lower than 5, the second guess should be 3. The strategy is to narrow down to the answer by partitioning the available range into two equal parts at every turn.

The successive approximation method works similarly in that the MSB is used to partition the original input range of an ADC into halves, the second MSB divides the remaining half into two quarters of the input range, and so forth. Figure 4.12a shows the architecture of this type of converter. The advantage of this technique is that the conversion time is uniform for any input voltage large or small, but the disadvantage of the technology is the use of complex hardware for implementation. The ATmega164 has a 10-bit successive approximation analog-to-digital converter.

4.4.2 INTEGRATION

The integration technique uses an integrator, a comparator, and a controller to convert analog signals to digital signals. A sampled analog signal is integrated over a fixed period, say n clock cycles of the digital system. Another fixed reference signal is integrated over time and compared with the input analog signal integrated. Although the value of the reference signal integrated is smaller than the input analog signal integrated, the reference signal is continuously integrated,



(a) Successive-Approximation Converter

(b) Integration-based Converter

(c) Counter-based Converter

(d) Parallel Converter

Figure 4.12: Four different technologies used to convert analog signals to digital signals: (a) successive approximation converter, (b) integration-based converter, (c) counter-based converter, and (d) parallel ADC.

and the time for the integration is measured. When the two integrated values equal, the measured time is converted to a digital encoded value. Figure 4.12b shows the system components of the converter. One disadvantage of this technique is the varying time for the conversion process. A small analog value will take less time to convert compared with a large value.

4.4.3 COUNTER-BASED CONVERSION

The third technique to convert an analog signal to a digital signal is the counter-based conversion. This conversion is performed with the help of a counter, a DAC, and a comparator. The counter starts at 0 and counts up. As the counter counts up, the corresponding value is converted to an analog value and compared with an input analog signal. As long as the input analog signal is greater than the signal generated by the DAC, the counter counts up and the process continues. When the comparator detects that the signal from the DAC is greater than the input analog signal, the counter value is then converted to a digital value representing the sampled analog signal. Figure 4.12c shows the overall architecture of this converter. Similar to the converter based on the integration principle, the disadvantage of this type of converter is the varying conversion time.

4.4.4 PARALLEL CONVERSION

The last technique allows the quickest conversion time among the techniques we discussed. A parallel converter uses a large number of comparators and circuitry to simultaneously measure the input signal and convert it to a digital value. The obvious disadvantage of this technique is the cost involved in building the circuitry. Figure 4.12d shows the architecture of the converter.

Example: An LED array is used to display the magnitude of an analog DC voltage as shown in Figure 4.13. A series of analog op amp comparator circuits are used to illuminate corresponding LEDs. The comparator circuits form a parallel analog-to-digital converter. The interface between the comparators and the 10 mm LEDs will be discussed in an upcoming chapter. Each comparator circuit is set with a different threshold voltage using the series resistor network. As the magnitude of the input voltage increases more LEDs are illuminated.

4.5 THE ATMEL ATMEGA164 ADC SYSTEM

The Microchip ATmega164 microcontroller is equipped with a flexible and powerful ADC system. It has the following features:

- 10-bit resolution,
- ±2 LSB absolute accuracy,
- 13 ADC clock cycle conversion time,
- 8 multiplexed single-ended input channels,

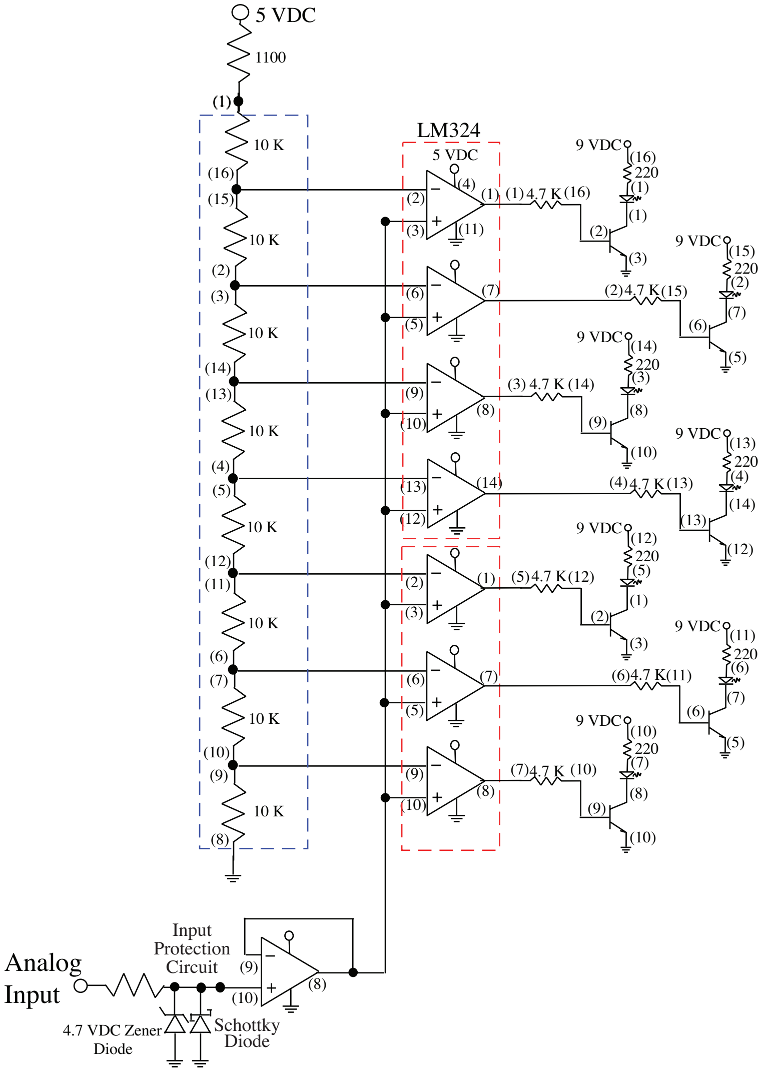


Figure 4.13: Parallel ADC converter.

- selectable right or left result justification, and
- 0 to Vcc ADC input voltage range.

The first feature of discussion is "10-bit resolution." Resolution is defined as:

$$\text{resolution} = (V_{RH} - V_{RL})/2^b.$$

V_{RH} and V_{RL} are the ADC high and low reference voltages, whereas b is the number of bits available for conversion. For the ATmega164 with reference voltages of 5 VDC, 0 VDC, and 10 bits available for conversion, resolution is 4.88 mV. Absolute accuracy specified as ±2 LSB is then ±9.76 mV at this resolution.

It requires 13 analog-to-digital clock cycles to perform an analog-to-digital conversion. The ADC system may be operated at a slower clock frequency (50–200 kHz) than the main microcontroller clock source. The main microcontroller clock is divided down using the ADC Prescaler Select (ADPS[2:0]) bits in the ADC Control and Status Register A (ADCSRA).

The ADC is equipped with a single successive approximation converter. Only a single ADC channel may be converted at a given time. The input of the ADC is equipped with an eight-input analog multiplexer. The analog input for conversion is selected using the MUX[4:0] bits in the ADC Multiplexer Selection Register (ADMUX).

The 10-bit result from the conversion process is placed in the ADC Data Registers, ADCH and ADCL. These two registers provide 16 bits of space for the 10-bit result. The result may be left justified by setting the ADLAR (ADC Left Adjust Result) bit of the ADMUX register. Right justification is provided by clearing this bit.

The analog input voltage for conversion must be between 0 and Vcc V. If this is not the case, external circuitry must be used to ensure the analog input voltage is within these prescribed bounds as discussed earlier in the chapter.

4.5.1 BLOCK DIAGRAM

The block diagram for the ATmega164 ADC conversion system is shown in Figure 4.14. The left edge of the diagram provides the external microcontroller pins to gain access to the ADC. The eight analog input channels are provided at ADC[7:0], and the ADC reference voltage pins are provided at AREF and AVCC. The key features and registers of the ADC system previously discussed are included in the diagram.

4.5.2 REGISTERS

The key registers for the ADC system are shown in Figure 4.15. It must be emphasized that the ADC system has many advanced capabilities that we do not discuss here. Our goal is to review the basic ADC conversion features of this powerful system. We have already discussed many of the register settings already. We will discuss each register next.

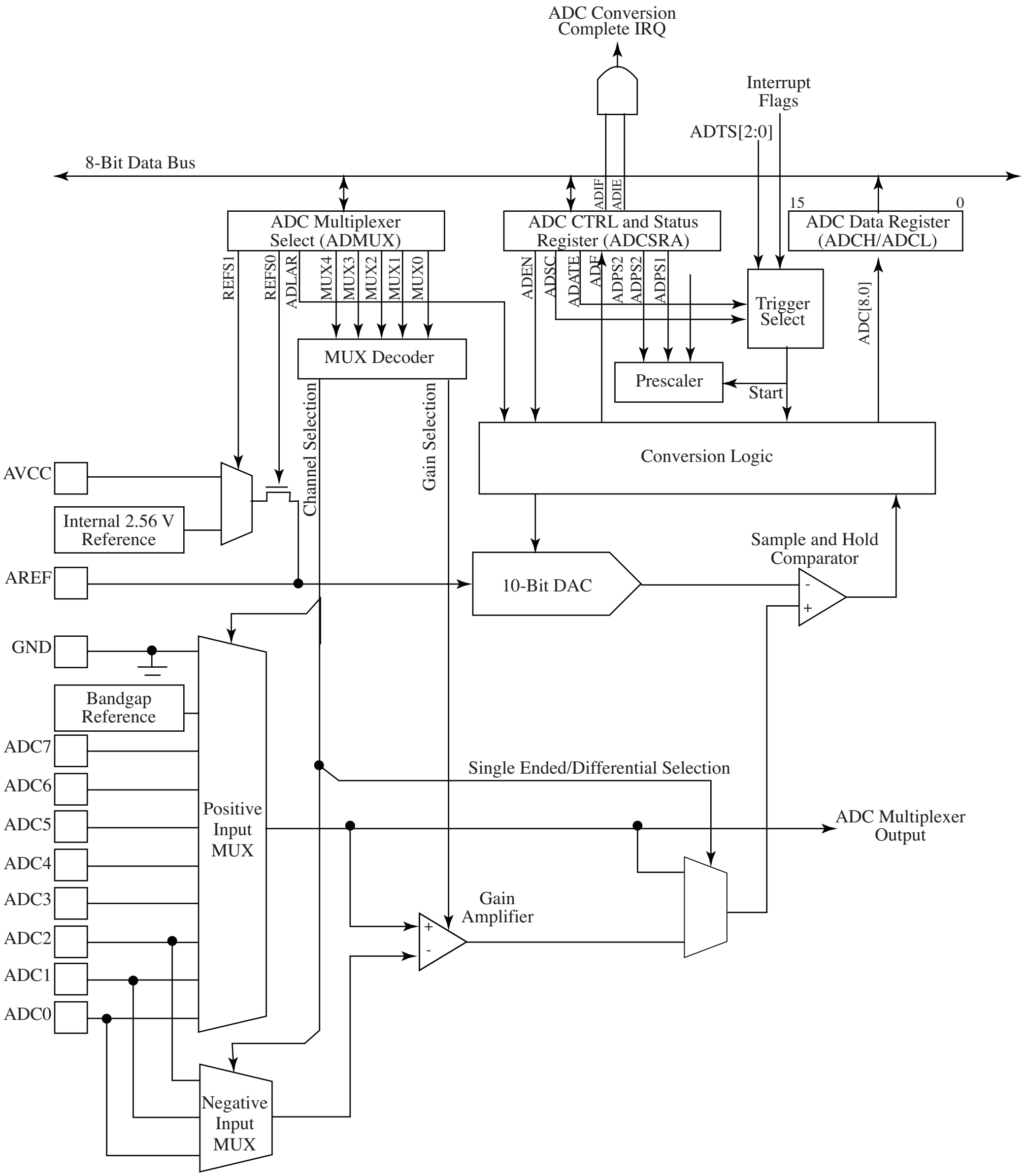


Figure 4.14: Microchip AVR ATmega164 ADC block diagram. (Figure used with permission of Microchip Technology, Inc. (www.microchip.com). All rights reserved.)

ADC Multiplexer Selection Register - ADMUX

REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
7							0

ADC Control and Status Register A - ADCSRA

ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
7							0

ADC Data Register - ADCH and ADCL (ADLAR = 0)

---	---	---	---	---	---	ADC9	ADC8	ADCH
15							8	
ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
7							0	

ADC Data Register - ADCH and ADCL (ADLAR = 1)

ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
15							8	
ADC1	ADC0	---	---	---	---	---	---	ADCL
7							0	

Figure 4.15: ADC registers.

ADC Multiplexer Selection Register

As previously discussed, the ADMUX register contains the ADLAR bit to select left or right justification and the MUX[4:0] bits to determine which analog input will be provided to the ADC for conversion. To select a specific input for conversion is accomplished when a binary equivalent value is loaded into the MUX[4:0] bits. For example, to convert channel ADC7, "00111" is loaded into the ADMUX register. This may be accomplished using the following C instruction:

```
ADMUX = 0x07;
```

The REFS[1:0] bits of the ADMUX register are also used to determine the reference voltage source for the ADC system. These bits may be set to the following values:

- REFS[0:0] = 00: AREF used for ADC voltage reference,
- REFS[0:1] = 01: AVCC with external capacitor at the AREF pin,
- REFS[1:0] = 10: reserved, and

- REFS[1:1] = 11: internal 2.56-VDC voltage reference with an external capacitor at the AREF pin.

ADC Control and Status Register A

The ADCSRA register contains the ADC Enable (ADEN) bit. This bit is the "on/off" switch for the ADC system. The ADC is turned on by setting this bit to logic high. The ADC Start Conversion (ADSC) bit is also contained in the ADCSRA register. Setting this bit to logic 1 initiates an ADC. The ADCSRA register also contains the ADC Interrupt flag (ADIF) bit. This bit sets to logic 1 when the ADC is complete. The ADIF bit is reset by writing a logic 1 to this bit.

The ADPS[2:0] bits are used to set the ADC clock frequency. The ADC clock is derived from dividing down the main microcontroller clock. The ADPS[2:0] may be set to the following values.

- ADPS[2:0] = 000: division factor: 2
- ADPS[2:0] = 001: division factor: 2
- ADPS[2:0] = 010: division factor: 4
- ADPS[2:0] = 011: division factor: 8
- ADPS[2:0] = 100: division factor: 16
- ADPS[2:0] = 101: division factor: 32
- ADPS[2:0] = 110: division factor: 64
- ADPS[2:0] = 111: division factor: 128

ADC Data Registers (ADCH and ADCL)

As previously discussed, the ADC Data Register contains the result of the ADC. The results may be left (ADLAR=1) or right (ADLAR=0) justified.

4.5.3 PROGRAMMING THE ADC

Provided below are two functions to operate the ATmega164 ADC system. The first function "InitADC( )" initializes the ADC by first performing a dummy conversion on channel 0. In this particular example the ADC prescalar is set to 8 (the main microcontroller clock was operating at 10 MHz).

The function then enters a while loop waiting for the ADIF bit to set, indicating the conversion is complete. After conversion, the ADIF bit is reset by writing a logic 1 to it.

The second function "ReadADC(unsigned char)" is used to read the analog voltage from the specified ADC channel. The desired channel for conversion is passed in as an unsigned character variable. The result is returned as a left-justified, 10-bit binary result. The ADC prescalar must be set to 8 to slow down the ADC clock at higher external clock frequencies (10 MHz) to obtain accurate results. After the ADC is complete, the results in the 8-bit ADCL and ADCH result registers are concatenated into a 16-bit unsigned integer variable and returned to the function call. In the final chapter of the book, we show how to convert an unsigned integer into a floating point voltage that may be displayed on a LCD.

```
//*************************************************************
//InitADC: initialize analog-to-digital converter
//*************************************************************

void InitADC( void)
{
ADMUX = 0;                                      //Select channel 0
ADCSRA = 0xC3;                                  //Enable ADC & start 1st
                                                //dummy
                                                //conversion
                                                //Set ADC module prescalar
                                                //to 8 critical for
                                                //accurate ADC results
while (!(ADCSRA & 0x10));                       //Check if conversation is
                                                //ready
ADCSRA |= 0x10;                                 //Clear conv rdy flag -
                                                //set the bit
}

//*************************************************************
//ReadADC: read analog voltage from analog-to-digital converter -
//the desired channel for conversion is passed in as an unsigned
//character variable. The result is returned as a left justified,
//10 bit binary result. The ADC prescalar must be set to 8 to
//slow down the ADC clock at higher external clock frequencies
//(10 MHz) to obtain accurate results.
//*************************************************************

unsigned int ReadADC(unsigned char channel)
{
unsigned int binary_weighted_voltage, binary_weighted_voltage_low;
unsigned int binary_weighted_voltage_high; //weighted binary
                                           //voltage

ADMUX = channel;                      //Select channel
ADCSRA |= 0x43;                       //Start conversion
                                      //Set ADC module prescalar
                                      //to 8 critical for
                                      //accurate ADC results
while (!(ADCSRA & 0x10));             //Check if converstion is
                                      //ready
ADCSRA |= 0x10;                       //Clear Conv rdy flag - set
                                      //the bit
binary_weighted_voltage_low = ADCL;   //Read 8 low bits first
                                      //(important)
                                      //Read 2 high bits,
                                      //multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage =
            = binary_weighted_voltage_low | binary_weighted_
              voltage_high;
return binary_weighted_voltage;       //ADCH:ADCL
}

//*************************************************************
```

Example: Often it is required to display the result of an ADC conversion on an LCD display. The ADC result is often a binary weighted voltage. To display the voltage on an LCD display, each portion of the result must be sent to the display as an ASCII value. The following function takes the binary weighted result from the ADC, scales it to a corresponding floating point value between 0 and 5 VDC, and then converts the individual digits to ASCII.

```
//***************************************************************
//convert_display_voltage_LCD: converts binary weighted voltage
//to ASCII representation and prints result to LCD screen
//***************************************************************
void convert_display_voltage_LCD(int binary_voltage)
{
float actual_voltage;                  //voltage 0 to 5 VDC
int   all_integer_voltage;             //integer representation
                                       //of voltage
                                       //int representation of
                                       //voltage
int   hundreths_place, tens_place, ones_place;
                                       //char representation of
                                       //voltage
char  hundreths_place_char, tens_place_char, ones_place_char;

                                        // display analog voltage
                                        //on LCD
putcommand(0xC0);                       //LCD cursor to line 2

                                        //scale float voltage
                                        //0..5V
actual_voltage = ((float)(binary_voltage)/(float)(0x3FF))*5.0;
                                        //voltage represented 0
                                        //to 500
all_integer_voltage=actual_voltage * 100;//represent as all
                                         //integer
                                        //convert to ASCII
hundreths_place = all_integer_voltage/100;//isolate first digit
hundreths_place_char = (char)(hundreths_place + 48);
putchar(hundreths_place_char);          //display first digit
putchar('.');                           //print decimal point to
                                        //LCD
                                        //isolate tens place
xtens_place = (int)((all_integer_voltage - (hundreths_place*100))/
                                          10);
tens_place_char=(char)(tens_place+48);  //convert to ASCII
putchar(tens_place_char);               //print to LCD
                                        //isolate ones place
ones_place = (int)((all_integer_voltage - (hundreths_place*100))
ones_place_char=(char)(ones_place+48);  //convert to ASCII
putchar(ones_place_char);               //print to LCD
putchar('V');                           //print unit V
}

//***************************************************************
```

4.5.4 DIGITAL-TO-ANALOG CONVERSION

Once an analog signal is converted to a digital representation with the help of the ADC process, frequently, the signal is processed further. The modified signal is then converted back to another analog signal. A simple example of such a conversion occurs in digital audio processing. Human voice is converted to a digital signal, modified, processed, and converted back to an analog signal for people to hear. The process to convert digital signals to analog signals is completed by a digital-to-analog converter (DAC). The most commonly used technique to convert digital signals to analog signals is the summation method shown in Figure 4.16.

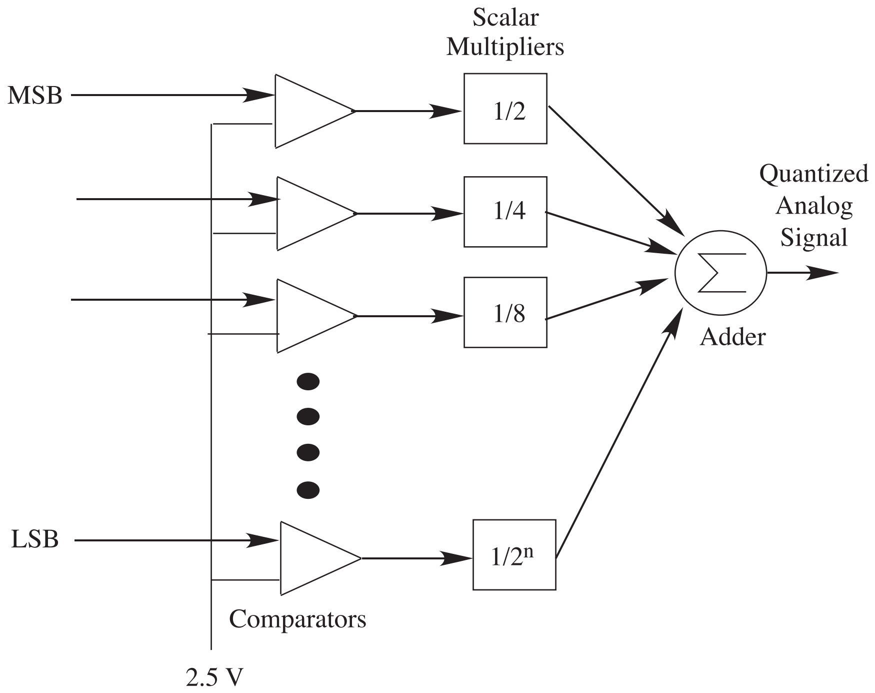


Figure 4.16: A summation method to convert a digital signal into a quantized analog signal. Comparators are used to clean up incoming signals, and the resulting values are multiplied by a scalar multiplier, and the results are added to generate the output signal. For the final analog signal, the quantized analog signal should be connected to a low pass filter followed by a transducer interface circuit.

With the summation method of DAC, a digital signal, represented by a set of 1's and 0's, enters the DAC from the MSB to the LSB. For each bit, a comparator checks its logic state, high or low, to produce a clean digital bit, represented by a voltage level. Typically, in a microcontroller context, the voltage level is 5 or 0 V to represent logic 1 or logic 0, respectively. The voltage is then multiplied by a scalar value based on its significant position of the digital signal as shown in Figure 4.16. Once all bits for the signal have been processed, the resulting voltage levels are summed to produce the final analog voltage value. Notice that the production of a desired analog signal may involve further signal conditioning such as a low pass filter to "smooth" the quantized analog signal and a transducer interface circuit to match the output of the DAC to the input of an output transducer.

Most microcontrollers are not equipped with an onboard DAC system. Instead, a DAC may be connected to the processor via a parallel or serial connection as illustrated in the following examples.

Example: Provided in Figure 4.17 is a parallel configured DAC. Analog signals may be reconstructed by sequentially sending the corresponding digital sample points. The analog signal consists of a series of these points. In this example, a ramp waveform is produced. The ramp up and ramp down time are independently set using potentiometers attached to ATmega164 ADC inputs. The output from the DAC is sent to an op amp conditioning circuit to vary the amplitude and DC offset of the analog signal.

```
//**********************************************************************
//DAC.c
//**********************************************************************
//Pin 21 PD7 DAC0808 A1(5)
//Pin 20 PD6 DAC0808 A2(6)
//Pin 19 PD5 DAC0808 A3(7)
//Pin 18 PD4 DAC0808 A4(8)
//Pin 17 PD3 DAC0808 A5(9)
//Pin 16 PD2 DAC0808 A6(10)
//Pin 15 PD1 DAC0808 A7(11)
//Pin 14 PD0 DAC0808 A8(12)
//
//Pin 40 PA0 analog input - fall time
//Pin 39 PC1 analog input - rise time
//**********************************************************************
//include files*********************************************************
//MICROCHIP register definitions for ATmega164
//**********************************************************************
//include files*********************************************************
//MICROCHIP register definitions for ATmega164
//Include Files: choose the appropriate include file depending on
//the compiler in use - comment out the include file not in use.

//include file(s) for JumpStart C for AVR Compiler*****************
#include<iom164pv.h>                   //contains reg definitions
```

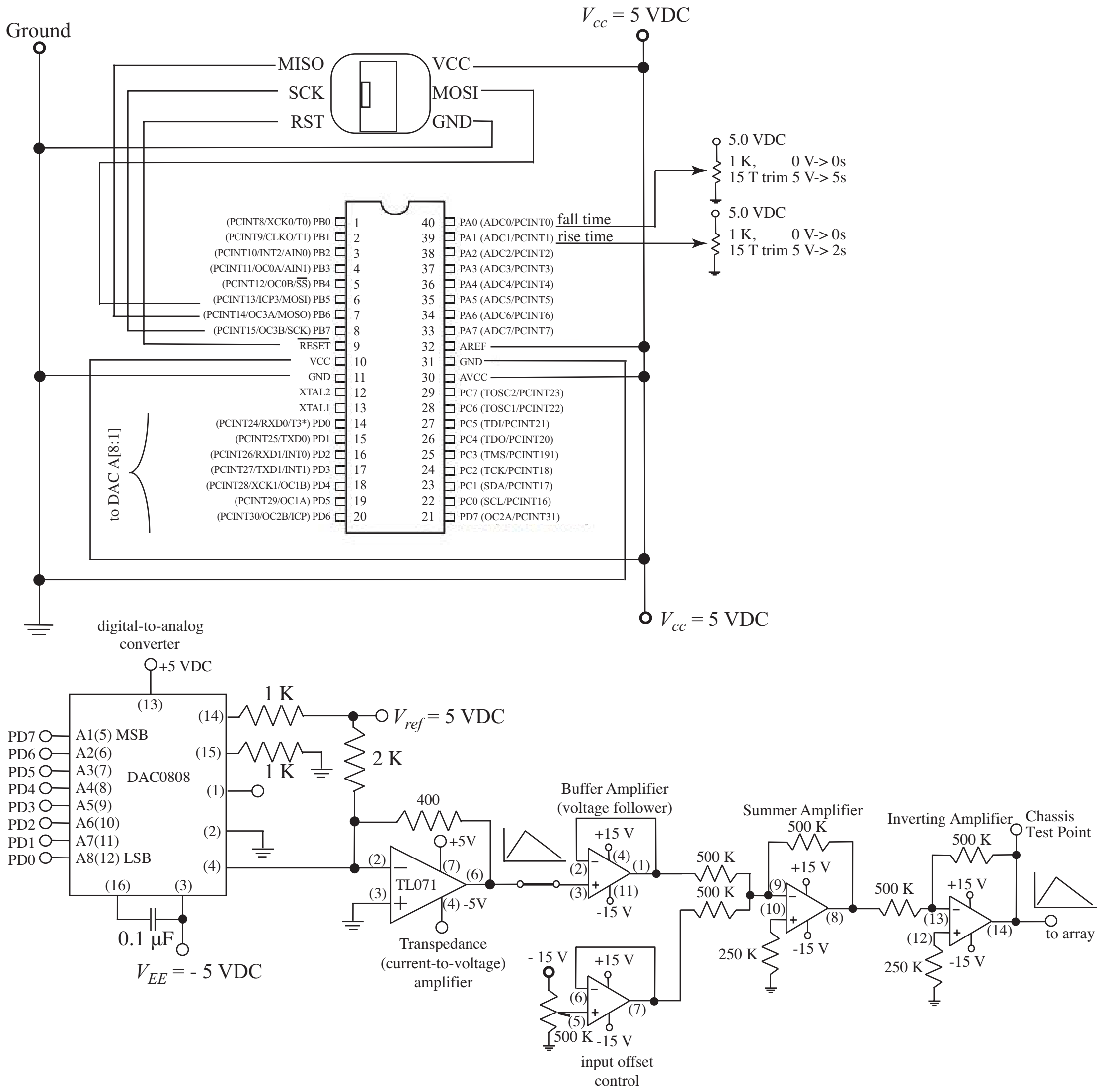


Figure 4.17: Parallel configured DAC.

```
//include file(s) for the Atmel Studio gcc compiler
//#include <avr/io.h>                  //contains reg definitions


//function prototypes*************************************************
                                          //delay number 65.5ms int
void delay(unsigned int number_of_65_5ms_interrupts);
void InitADC(void);                       //initialize ADC
void initialize_ports(void);              //initializes ports
void power_on_reset(void);                //returns to startup state
unsigned int  ReadADC(unsigned char chan);//read value from ADC results
void init_timer0_ovf_interrupt(void);     //initialize timer0 overflow
void read_rise_time(void);
void read_fall_time(void);
void generate_waveform_via_DAC(void);
                                          //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:19


//main program********************************************************

//global variables
unsigned int input_delay;
unsigned int rise_time_num_interrupts;
unsigned int fall_time_num_interrupts;



void main(void)
{
power_on_reset();                        //returns to startup condition

while(1)
  {
   read_rise_time();                     //determine rise time
   read_fall_time();                     //determine close time
   generate_waveform_via_DAC();          //generate waveform via DAC
   }
}//end main

//Function definitions
//**********************************************************************
//power_on_reset:
//**********************************************************************

void power_on_reset(void)
{
initialize_ports();                     //initialize ports
init_timer0_ovf_interrupt();            //init Timer0 to serve as elapsed
                                        //time "clock tick"
}

//**********************************************************************
//initialize_ports: provides initial configuration for I/O ports
//**********************************************************************

void initialize_ports(void)
{
//PORTA
DDRA=0xff;                              //PORTA[7:0] as output
PORTA=0x00;

//PORTB
DDRB=0xff;                              //PORTB[7:0] as output
PORTB=0x00;

//PORTC
DDRC=0xFC;                              //PORTC[7:2] output, [1:0] input
PORTC=0x00;

//PORTD
DDRD=0xff;                              //PORTD[7:0] output
PORTD=0x00;
}

//***********************************************************************
//***********************************************************************
```

```
//delay(unsigned int num_of_65_5ms_interrupts): this generic delay
//function provides the specified delay as the number of 65.5 ms "clock
//ticks" from the Timer0 interrupt.
//Note: this function is only valid when using a 1 MHz time base
//      RC oscillator set for 8 MHz and divide by 8 fuse set
//**********************************************************************

void delay(unsigned int number_of_65_5ms_interrupts)
{
TCNT0 = 0x00;                          //reset timer0
input_delay = 0;
while(input_delay <= number_of_65_5ms_interrupts)
  {
  ;
  }
}

//**********************************************************************
//InitADC: initialize ADC converter
//**********************************************************************

void InitADC(void)
{
ADMUX = 0;                             //Select channel 0
ADCSRA = 0xC6;                         //Enable ADC & start 1st dummy
                                       //conversion
                                       //Set ADC module prescalar to 64
                                       //critical for accurate results
while (!(ADCSRA & 0x10));              //Check if conversation is ready
ADCSRA |= 0x10;                        //Clear conv rdy flag-set the bit
}

//**********************************************************************
//ReadADC: read analog voltage from ADC - the desired channel for
//conversion is passed in as an unsigned character variable.  The result
//is returned as a left justified, 10 bit binary result.  The ADC
//prescalar must be set to 64 to slow down the ADC clock at higher clock
//frequencies (8 MHz) to obtain accurate results.  The ADC clock must
//operate at 200 kHz or less for accurate results.
```

```
//**********************************************************************

unsigned int ReadADC(unsigned char channel)
{
unsigned int binary_weighted_voltage, binary_weighted_voltage_low;
unsigned int binary_weighted_voltage_high; //weighted binary voltage

ADMUX = channel;                       //Select channel
ADCSRA |= 0x46;                        //Start conversion
                                       //Set ADC module prescalar to 64
     //critical for accurate results
while (!(ADCSRA & 0x10));              //Check if conversion is ready
ADCSRA |= 0x10;                        //Clr Conv rdy flag - set the bit
binary_weighted_voltage_low = ADCL;    //Read 8 low bits 1st (important)
                                       //Read 2 high bits,multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage=binary_weighted_voltage_low|
                        binary_weighted_voltage_high;
return binary_weighted_voltage;       //ADCH:ADCL
}

//*********************************************************************
//int_timer0_ovf_interrupt(): The Timer0 overflow interrupt is being
//employed as a time base for a master timer for this project. The
//internal RC oscillator operating at 1 MHz is divided by 256.  The
//8-bit Timer0 register (TCNT0) overflows every 256 counts, 65.5 ms.
//*********************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04; //divide timer0 timebase by 256, overflow every 65.5ms
TIMSK0 = 0x01; //enable timer0 overflow interrupt
asm("SEI");    //enable global interrupt
}

//*********************************************************************
//*********************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing the
```

```
//corresponding interrupt handling vector.
//*********************************************************************

void timer0_interrupt_isr(void)
{
input_delay++;
}

//*********************************************************************
//*********************************************************************
//read_rise_time:  0V:0s -> 5V:2s
//*********************************************************************

void read_rise_time(void)
{
float   rise_time;
unsigned int rise_time_setting;

InitADC();                            //Initialize ADC
rise_time_setting=ReadADC(0x00);      //rise time potentiometer
                                      //connected to ADC Ch 0 (0000_0000)
                                      //convert volt from binary to
                                      //floating point
rise_time = ((float)(rise_time_setting)/(float)(0x0400));
                                      //convert voltage to seconds
rise_time = ((rise_time * 2.0) + 0.0);
                                      //convert secs to timer interrupts
rise_time_num_interrupts=(unsigned int)(rise_time/0.0655);
}

//*********************************************************************
//*********************************************************************
//read_fall_time:  0V:0s -> 5V:5s
//*********************************************************************

void read_fall_time(void)
{
float   fall_time;
unsigned int fall_time_setting;
```

```
InitADC();                            //Initialize ADC
fall_time_setting=ReadADC(0x01);      //fall time potentiometer
                                      //connected to ADC Ch 1 (0000_0001)
                                      //convert volt from binary to
                                      //floating point
fall_time = ((float)(fall_time_setting)/(float)(0x0400));
                                      //convert voltage to seconds
fall_time = ((fall_time * 5.0) + 0.0);
                                      //convert secs to timer interrupts
fall_time_num_interrupts=(unsigned int)(fall_time/0.0655);
}

//*********************************************************************
//void generate_waveform_via_DAC(void)
//*********************************************************************

void generate_waveform_via_DAC(void)
{
unsigned char ramp_up_increment;
unsigned int  ramp_up_increment_int;
unsigned char ramp_down_increment;
unsigned int ramp_down_increment_int;
unsigned char DAC_output;
unsigned char  i;

DAC_output = 0;
PORTB = DAC_output;
                                      //ramp up - calc ramp up increment
if(rise_time_num_interrupts != 0)
  ramp_up_increment_int =
  (unsigned int)((200.0/(double)(rise_time_num_interrupts))+0.5);
else
  ramp_up_increment_int = 200;
ramp_up_increment = (unsigned char)(ramp_up_increment_int);

                                      //generate ramp up waveform
for(i=0; i<=rise_time_num_interrupts; i++)
  {
```

```
  delay(0);
  DAC_output = DAC_output + ramp_up_increment;
  if(DAC_output > 200) DAC_output = 200;
  PORTB = DAC_output;
  }

//DAC_output = 180;
//PORTB = DAC_output;
                                      //ramp up - calc ramp up increment
if(fall_time_num_interrupts != 0)
  ramp_down_increment_int =
  (unsigned int)((180.0/(double)(fall_time_num_interrupts))+0.5);
else
  ramp_down_increment_int = 180;
ramp_down_increment = (unsigned char)(ramp_down_increment_int);

                                      //generate ramp up waveform
for(i=0; i<=fall_time_num_interrupts; i++)
  {
  delay(0);
  DAC_output = DAC_output - ramp_down_increment;
  if(DAC_output < 0) DAC_output = 0;
  PORTB = DAC_output;
  }
}

//*************************************************************************
//end of file: knh_generator.c
//*************************************************************************
```

4.6 SUMMARY

In this chapter, we presented the differences between analog and digital signals and used this knowledge to discuss three subprocessing steps involved in ADCs: sampling, quantization, and encoding. We also presented the quantization errors and the data rate associated with the analog-to-digital process. The dynamic range of an ADC, one of the measures to describe a conversion process, was also presented. Two different categories exist to represent technologies used to convert analog signals to their corresponding digital signals: direct approach and indirect approaches. For the direct approach, we presented the successive approximation, counter-based, and parallel conversion techniques. For the indirect approach, the integration-based conversion technique was discussed. We then reviewed the operation, registers, and actions required to program the ADC system aboard the ATmega164. We concluded the chapter with a brief presentation of DACs.

4.7 REFERENCES AND FURTHER READING

Atmel 8-bit AVR Microcontroller with 16/32/64/128K Bytes In-System Programmable Flash, ATmega164A, ATmega164PA, ATmega324A, ATmega324PA, ATmega644A, ATmega644PA, ATmega1284, ATmega1284P, 8272C-AVR-06/11, data sheet: 8272C-AVR-06/11, Atmel, San Jose, CA, 2015. 116

Barrett, S. F. and Pack, D. J. *Microcontrollers Fundamentals for Engineers and Scientists*. Morgan & Claypool, San Rafael, CA, 2006. DOI: 10.2200/s00025ed1v01y200605dcs001. 103, 114

Barrett, S. F. and Pack, D. J. *Microcontroller Theory and Applications: HC12 and S12*, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 2008. 99

Enderle, J., Blanchard, S., and Bronzino, J. *Introduction to Biomedical Engineering*. Academic Press, San Diego, 2000. 109

Faulkenberry, L. *An Introduction to Operational Amplifiers*. John Wiley & Sons, New York, 1977. 110, 111, 112, 113

Oppenheim, A. and Schafer, R. *Discrete-Time Signal Processing*, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 1999. 107

Thomas, R. and Rosa, A. *The Analysis and Design of Linear Circuits*, 4th ed., Wiley & Sons, New York, 2003. 109

4.8 CHAPTER PROBLEMS

1. Given a sinusoidal signal with 500-Hz frequency, what should be the minimum sampling frequency for an ADC, if we want to faithfully reconstruct the analog signal after the conversion?

2. If 12 bits are used to quantize a sampled signal, what is the number of available quantized levels? What will be the resolution of such a system if the input range of the ADC is 10 V?

3. Given the 12-V input range of an ADC and the desired resolution of 0.125 V, what should be the minimum number of bits used for the conversion?

4. Perform a trade-off study on the four technologies used for the ADC. Use cost, conversion time, and accuracy as the list of criteria.

5. Investigate the ADCs in your audio system. Find the sampling rate, the quantization bits, and the technique used for the conversion.

6. A flex sensor provides 10 kΩ of resistance for 0° flexure and 40 kΩ of resistance for 90° of flexure. Design a circuit to convert the resistance change to a voltage change (hint: consider a voltage divider). Then design a transducer interface circuit to convert the output from the flex sensor circuit to voltages suitable for the ATmega164 ADC system.

7. If an analog signal is converted by an ADC to a binary representation and then back to an analog voltage using a DAC, will the original analog input voltage be the same as the resulting analog output voltage? Explain.

8. Derive each of the characteristic equations for the classic operation amplifier configurations provided in Figure 4.9.

9. If a resistor was connected between the non-inverting terminal and ground in the inverting amplifier configuration of Figure 4.9a, how would the characteristic equation change?

10. A photodiode provides a current proportional to the light impinging on its active area. What classic operational amplifier configuration should be used to convert the diode output to a voltage?

11. Does the time to convert an analog input signal to a digital representation vary in a successive-approximation converter relative to the magnitude of the input signal?

12. A sensor provides an analog output signal from 1–2 VDC. Design an interface circuit to expand the sensor output from 0–5 VDC.

CHAPTER 5

Interrupt Subsystem

Objectives: After reading this chapter, the reader should be able to:

- explain the functions of microcontroller interrupt features;
- describe the general microcontroller interrupt response process;
- describe the ATmega164 interrupt features; and
- properly configure and program an interrupt event for the ATmega164.

5.1 INTERRUPT THEORY

A microcontroller normally executes instructions in an orderly fetch-decode-execute sequence as dictated by a user-written program, as shown in Figure 5.1. However, the microcontroller must

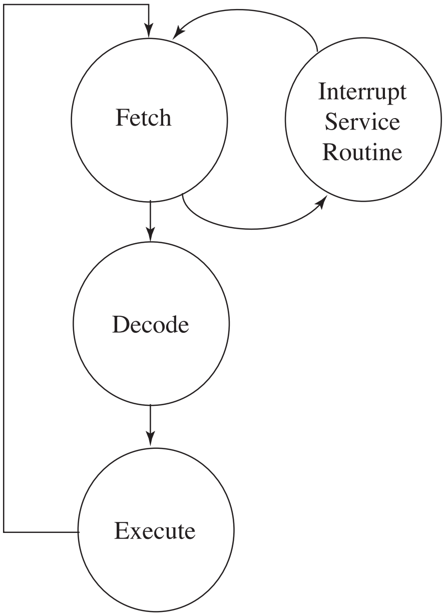


Figure 5.1: Microcontroller interrupt response.

be equipped to handle unscheduled, higher-priority events that might occur inside or outside the microcontroller. To process such events, a microcontroller requires an interrupt system [Barrett and Pack, 2006].

The interrupt system onboard a microcontroller allows it to respond to higher-priority events. These events are planned, but we do not know when they will occur. When an interrupt event occurs, the microcontroller will normally complete the instruction it is currently executing and then transition program control to interrupt event specific tasks. These tasks, which resolve the interrupt event, are organized into a function called an interrupt service routine (ISR). Each interrupt will normally have its own interrupt specific ISR. Once the ISR is completed, the microcontroller will resume processing where it left off before the interrupt event occurred, as shown in Figure 5.1.

5.2 ATMEGA164 INTERRUPT SYSTEM

The ATmega164 is equipped with powerful and flexible resources to respond to a complement of 31 interrupt sources. Three of the interrupts originate from external interrupt sources while the remaining 28 interrupts support the efficient operation of peripheral subsystems aboard the microcontroller. The ATmega164 interrupt sources are shown in Figure 5.2. The interrupts are listed in descending order of priority. As you can see the RESET has the highest priority, followed by the external interrupt request pins INT0 (pin 16), INT1 (pin 17), and INT2 (pin 3). The remaining interrupt sources are internal to the ATmega164. When interrupts are triggered simultaneously, the interrupt with the highest priority is serviced first and its corresponding interrupt service routine is executed.

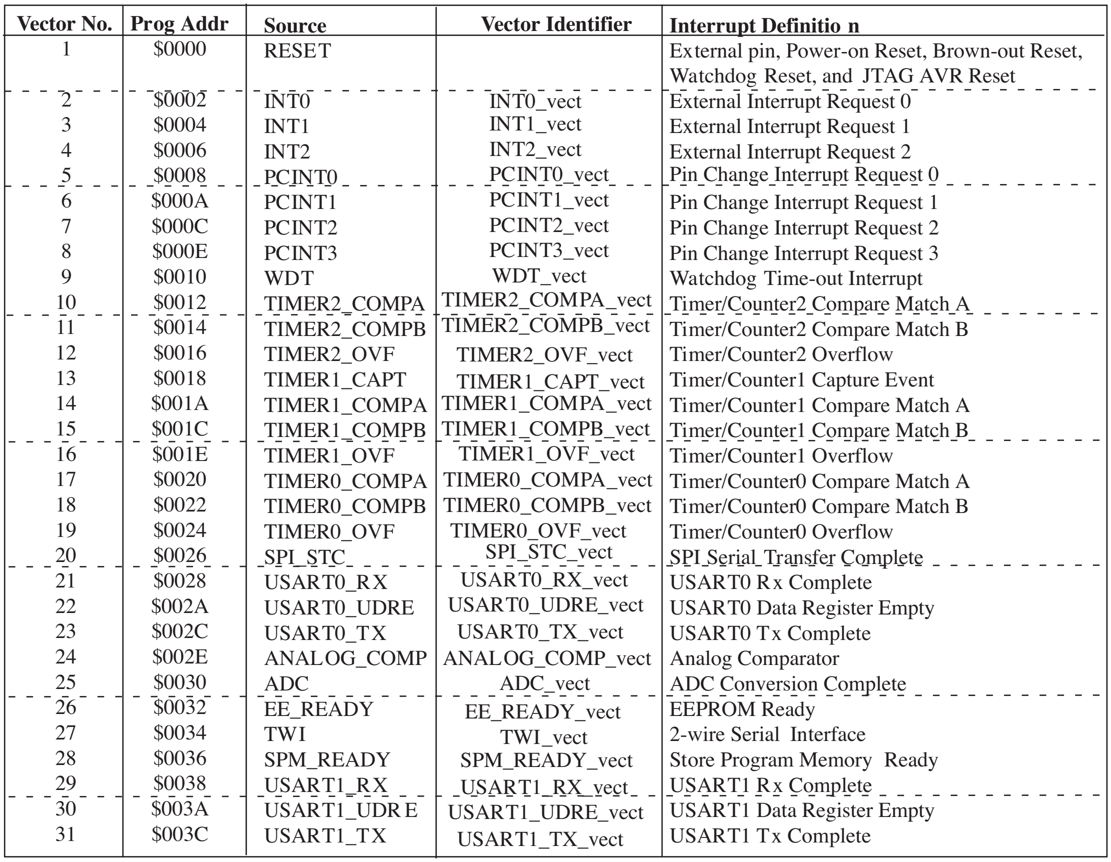
Most of the activities required to properly execute an interrupt are automatically accomplished by the microcontroller. When an interrupt occurs, the microcontroller completes the current instruction, stores key register values currently being used on the stack, places the beginning address of the interrupt service routine in the program counter, and starts executing instructions for the designated interrupt service routine (ISR) corresponding to the active interrupt source. It also turns off the interrupt system to prevent further interrupts while one is in progress. Execution of the ISR continues until the return from interrupt instruction (RETI) is encountered. Program control then reverts back to the main program.

The programmer's responsibility is to write an appropriate ISR corresponding to the desired response of the microcontroller in response to the interrupt, activate the interrupt system, and configure the interrupt system.

5.3 PROGRAMMING AN INTERRUPT

To program an interrupt, the user is responsible for the following actions.

- **Configuration.** Ensure the ISR for a specific interrupt is tied to the correct interrupt vector address, which points to the starting address of the ISR.



Vector No.	Prog Addr	Source	Vector Identifier	Interrupt Definitio n
1	$0000	RESET		External pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	INT0_vect	External Interrupt Request 0
3	$0004	INT1	INT1_vect	External Interrupt Request 1
4	$0006	INT2	INT2_vect	External Interrupt Request 2
5	$0008	PCINT0	PCINT0_vect	Pin Change Interrupt Request 0
6	$000A	PCINT1	PCINT1_vect	Pin Change Interrupt Request 1
7	$000C	PCINT2	PCINT2_vect	Pin Change Interrupt Request 2
8	$000E	PCINT3	PCINT3_vect	Pin Change Interrupt Request 3
9	$0010	WDT	WDT_vect	Watchdog Time-out Interrupt
10	$0012	TIMER2_COMPA	TIMER2_COMPA_vect	Timer/Counter2 Compare Match A
11	$0014	TIMER2_COMPB	TIMER2_COMPB_vect	Timer/Counter2 Compare Match B
12	$0016	TIMER2_OVF	TIMER2_OVF_vect	Timer/Counter2 Overflow
13	$0018	TIMER1_CAPT	TIMER1_CAPT_vect	Timer/Counter1 Capture Event
14	$001A	TIMER1_COMPA	TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
15	$001C	TIMER1_COMPB	TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
16	$001E	TIMER1_OVF	TIMER1_OVF_vect	Timer/Counter1 Overflow
17	$0020	TIMER0_COMPA	TIMER0_COMPA_vect	Timer/Counter0 Compare Match A
18	$0022	TIMER0_COMPB	TIMER0_COMPB_vect	Timer/Counter0 Compare Match B
19	$0024	TIMER0_OVF	TIMER0_OVF_vect	Timer/Counter0 Overflow
20	$0026	SPI_STC	SPI_STC_vect	SPI Serial Transfer Complete
21	$0028	USART0_RX	USART0_RX_vect	USART0 Rx Complete
22	$002A	USART0_UDRE	USART0_UDRE_vect	USART0 Data Register Empty
23	$002C	USART0_TX	USART0_TX_vect	USART0 Tx Complete
24	$002E	ANALOG_COMP	ANALOG_COMP_vect	Analog Comparator
25	$0030	ADC	ADC_vect	ADC Conversion Complete
26	$0032	EE_READY	EE_READY_vect	EEPROM Ready
27	$0034	TWI	TWI_vect	2-wire Serial Interface
28	$0036	SPM_READY	SPM_READY_vect	Store Program Memory Ready
29	$0038	USART1_RX	USART1_RX_vect	USART1 Rx Complete
30	$003A	USART1_UDR E	USART1_UDRE_vect	USART1 Data Register Empty
31	$003C	USART1_TX	USART1_TX_vect	USART1 Tx Complete

Figure 5.2: Microchip AVR ATmega164 interrupts. (Adapted from figure used with permission of Microchip Technology, Inc. (www.microchip.com). All rights reserved.)

- **Configuration.** Ensure the registers associated with the specific interrupt have been configured correctly.

- **Activation.** Ensure the interrupt system has been globally enabled. This is accomplished with the assembly language instruction SEI.

- **Activation.** Ensure the specific interrupt subsystem has been locally enabled.

- **Program.** Write the corresponding ISR.

5.4 APPLICATION

In this section we provide several templates for configuring interrupts using both the Atmel AVR Visual Studio gcc compiler and the ImageCraft JumpStart C for AVR compiler. We then provide several examples to illustrate how the templates are used to configure specific interrupts.

5.4.1 ATMEL AVR VISUAL STUDIO GCC COMPILER INTERRUPT TEMPLATE

The Atmel AVR Visual Studio gcc compiler uses a standardized naming convention to link an interrupt service routine to the correct interrupt vector address. The names for each interrupt service routine vector are provided in the fourth column of Figure 5.2. Also, the file containing interrupt definitions (interrupt.h) must be added to the include file list. The interrupt service routine definition begins with the keyword "ISR" followed by the specific name for the desired interrupt. The body of the interrupt service routine contains interrupt specific actions.

```
//********************************************************************
//include files
//Microchip register definitions for ATmega164
//
#include <avr/io.h>
#include <avr/interrupt.h>

//********************************************************************
//interrupt service routine definition
//********************************************************************

ISR(vector_identifier)
{

:
//programmer written interrupt specific actions
:

}

//********************************************************************
```

5.4.2 IMAGECRAFT JUMPSTART C FOR AVR COMPILER INTERRUPT TEMPLATE

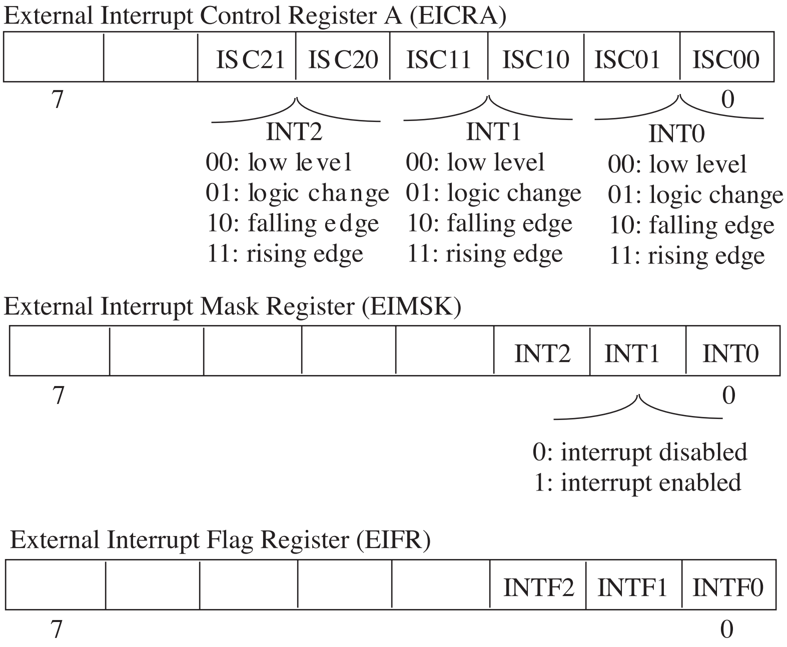
The ImageCraft JumpStart C for AVR compiler uses interrupt specific numbers to link an interrupt service routine to the correct interrupt vector address. The #pragma with the reserved word **interrupt_handler** is used to communicate to the compiler that the routine name that follows is an interrupt service routine. The number that follows the ISR name corresponds to the interrupt vector number in the first column of Figure 5.2. It is important that the ISR name used in the #pragma instruction matches the name of the ISR in the function body. Since the compiler knows the function is an ISR it will automatically place the RETI instruction at the end of the ISR.

```
//*****************************************************************
// ImageCraft JumpStart C for AVR compiler interrupt configuration
//*****************************************************************

//include file(s) for JumpStart C for AVR Compiler
#include<iom164pv.h>                   //contains reg definitions

#pragma interrupt_handler timer_handler:19

void timer_handler(void)
{

:
//programmer written interrupt specific actions
:


}
//*****************************************************************
```

5.4.3 EXTERNAL INTERRUPT EXAMPLE USING THE ATMEL AVR VISUAL STUDIO GCC COMPILER

The external interrupts INT0 (pin 16), INT1 (pin 17), and INT2 (pin 3) trigger an interrupt within the ATmega164 when a user-specified external event occurs at the pin associated with the specific interrupt. Interrupts INT0 and INT1 may be triggered with a level or an edge signal, whereas interrupt INT2 is edge-triggered only. The specific settings for each interrupt is provided in Figure 5.3.

In this example an interrupt will occur when a positive edge transition occurs on the ATmega164 INT0 external interrupt pin (PORTD[2]). The INT0 pin has a normally low mo-



Notes:
- INTFx flag sets when corresponding interrupt occurs.
- INTFx flag reset by executing ISR or writing logic one to flag.

Figure 5.3: Interrupt INT0, INT1, and INT2 registers.

mentary contact switch configuration. When the switch is depressed, an active low signal occurs on PORTD[2] which triggers the INT0 interrupt. When the interrupt occurs, the microcontroller services the interrupt service routine (ISR) for INT0. The INT0 ISR flashes the LED connected to PORTA[0]. The circuit diagram is provided in Figure 5.4.

```
//*************************************************************
//int0.c
//*************************************************************

#include <avr/io.h>
#include <avr/interrupt.h>

//function prototypes*****************************************
void initialize_ports(void);         //initializes ports
void initialize_interrupt0(void);    //initialize INT0
void delay_100ms(void);              //delay 100 ms
```

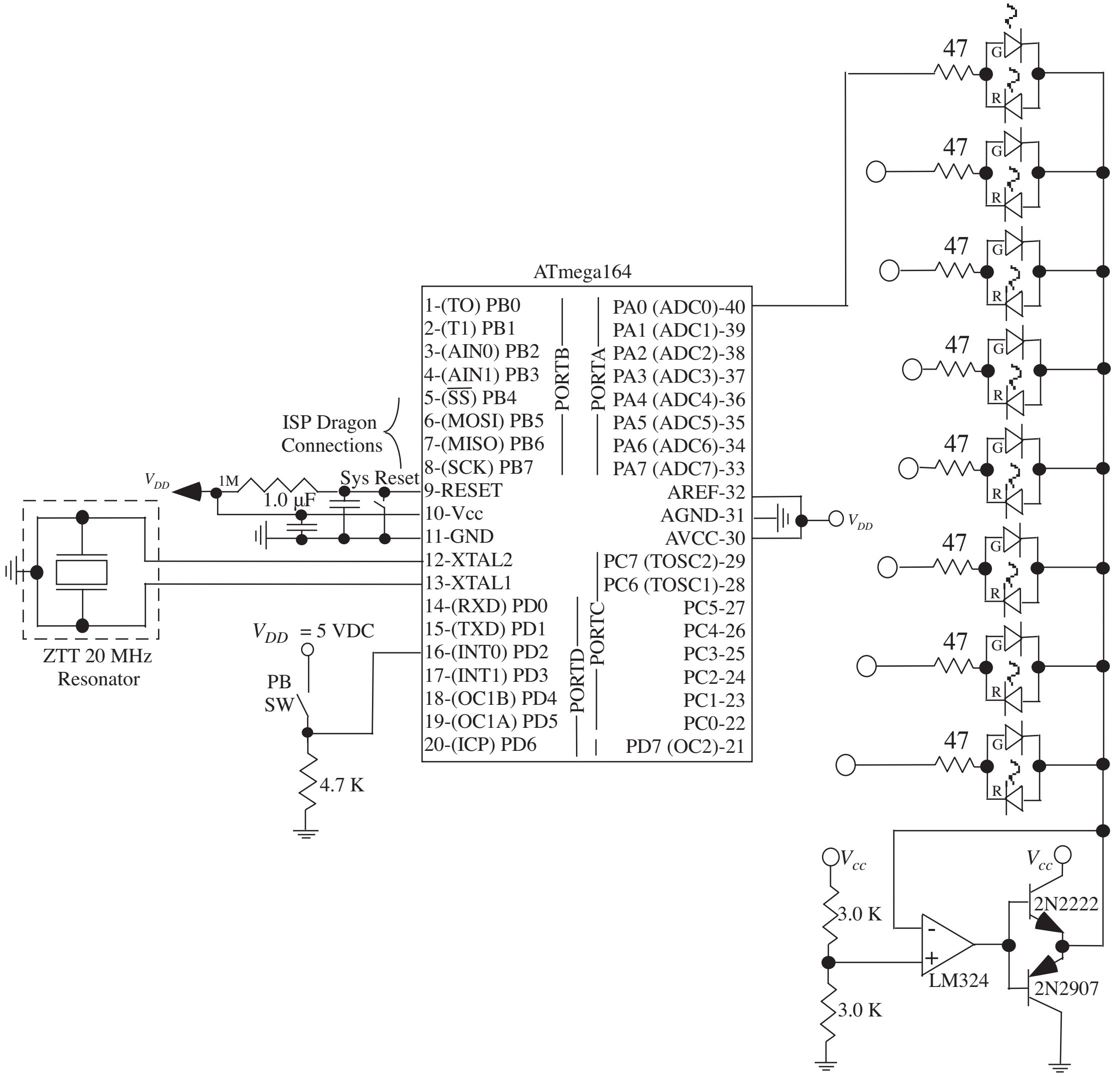


Figure 5.4: INT0 external interrupt.

```
//main program****************************************************

int main(void)
{
initialize_ports();                 //initialize ports
initialize_interrupt0();            //initialize INT0
PORTA = PORTA | 0x01;               //PORTA[0] high

while(1)
  {
  PORTA = PORTA | 0x01;             //PORTA[0] high
  }
}

//****************************************************************
//initialize_ports: provides initial configuration for I/O ports
//****************************************************************

void initialize_ports(void)
{
DDRA =0xff;  //set PORTA as output
PORTA=0x01;  //initialize low

DDRB =0x0a;  //PORTB[7:4] as input, set PORTB[3:0] as output
PORTB=0x00;  //initialize low

DDRC =0xff; //set PORTC as output
PORTC=0x00; //initialize low

DDRD =0xfb; //set PORTD as output, PORTD[2] as input 1111_1011
PORTD=0x00; //initialize low
}

//****************************************************************
//initialize_interrupt0: initializes interrupt INT0.
//Note: stack is automatically initialized by the compiler
//****************************************************************
```

```
void initialize_interrupt0(void) //initialize interrupt 0
{
DDRD = 0xFB;   //set PD2 (int0) as input
PORTD &=~0x04; //disable pullup resistor PD2
EIMSK = 0x01;  //enable int 0
EICRA = 0x03;  //set for positive edge trigger
asm("SEI");    //global interrupt enable
}

//*****************************************************************
//INT0 interrupt service routine definition
//*****************************************************************

ISR(INT0_vect)
{
delay_100ms();
PORTA = PORTA & 0xFE;              //PORTA[0] low

delay_100ms();
PORTA = PORTA | 0x01;              //PORTD[0] high

delay_100ms();
PORTA = PORTA & 0xFE;              //PORTA[0] low

delay_100ms();
PORTA = PORTA | 0x01;              //PORTD[0] high

delay_100ms();
PORTA = PORTA & 0xFE;              //PORTA[0] low

delay_100ms();
PORTA = PORTA | 0x01;              //PORTD[0] high

delay_100ms();
PORTA = PORTA & 0xFE;              //PORTA[0] low
}

//*****************************************************************
//*****************************************************************
```

```
//delay_100ms: inaccurate, yet simple method of creating delay
// - processor clock: ceramic resonator at 20 MHz
// - 100 ms delay requires 2M clock cycles
// - nop instruction requires 1 clock cycle to execute
//***************************************************************

void delay_100ms(void)
{
unsigned int i,j;
for(i=0; i < 2000; i++)
{
for(j=0; j < 1000; j++)
{
            asm("nop");            //inline assembly
}                       //nop: no operation
}                            //requires 1 clock cycle
}

//***************************************************************
```

5.4.4 AN INTERNAL INTERRUPT EXAMPLE USING THE JUMPSTART C FOR AVR COMPILER

In this example we use the Timer/Counter0 overflow interrupt system to provide prescribed delays within our program. Timer/Counter0 is an eight bit timer. It rolls over every time it receives 256 timer clock "ticks." There is an interrupt associated with the Timer/Counter0 overflow. If activated, the interrupt will occur every time the contents of the Timer/Counter0 transitions from 255 back to 0 count. We can use this overflow interrupt as a method of keeping track of real clock time (hours, minutes, and seconds) within a program. In this specific example, we use the overflow to provide precision program delays.

For this example, we assume the ATmega164 is being externally clocked by a 10 MHz ceramic resonator. The resonator frequency is further divided by 256 using the clock select bits CS[2:1:0] in the Timer/Counter Control Register B (TCCR0B). When CS[2:1:0] are set for [1:0:0], the incoming clock source is divided by 256. This provides a clock "tick" to Timer/Counter0 every 25.6 μs. Therefore, the eight bit Timer/Counter0 will rollover every 256 clock "ticks" or every 6.55 ms.

To create a precision delay, we write a function called delay. The function requires an unsigned integer parameter value indicating how many 6.55 ms interrupts should be used to specify the amount of delay needed. The function stays within a while loop until the desired number of interrupts has occurred. For example, to delay 1 s the function would be called with the parameter value "153." That is, it requires 153 interrupts occurring at 6.55 ms intervals to generate a 1 s delay.

The code snapshots to configure the Time/Counter0 Overflow interrupt is provided below, along with the associated interrupt service routine and the delay function.

```
//********************************************************************
//flash_led.c
//********************************************************************

#include <iom164pv.h>

//function prototypes
void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);
void timer0_interrupt_isr(void);
void delay(unsigned int number_of_6_55ms_interrupts);
void initialize_ports(void);

//initialize timer0 overflow interrupt
                                       //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:19


//global variables
unsigned int   input_delay;            //counts num of Timer/Counter0
                                       //Overflow interrupts

//main program

void main(void)
{
init_timer0_ovf_interrupt();  //initialize Timer/Counter0 Overflow
                              //interrupt - call once at beginning
initialize_ports();
PORTA = PORTA | 0x01;         //PORTD[0] high

while(1)
  {
  delay(153);                 //1 second delay
  PORTA = PORTA & 0xFE;       //PORTA[0] low
  delay(153);                 //1 second delay
  PORTA = PORTA | 0x01;       //PORTD[1] high
  }
}

//**********************************************************************
//int_timer0_ovf_interrupt(): The Timer/Counter0 Overflow interrupt is
//being employed as a time base for a master timer for this project. The
//ceramic resonator  operating at 10 MHz is divided by 256.  The 8-bit
//Timer0 register (TCNT0) overflows every 256 counts or every 6.55 ms.
//**********************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04;                //divide timer0 timebase by 256
                              //overflow occurs every 6.55ms
TIMSK0 = 0x01;                //enable timer0 overflow interrupt
asm("SEI");                   //enable global interrupt
}

//*********************************************************************
//*********************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing the
//corresponding interrupt handling vector.
//*********************************************************************

void timer0_interrupt_isr(void)
{
input_delay++;                         //increment overflow counter
}

//*********************************************************************
//delay(unsigned int num_of_6_55ms_interrupts): this generic delay
//function provides the specified delay as the number of 6.55 ms
//"clock ticks" generated from the Timer/Counter0 Overflow interrupt.
//Note: this function is only valid when using a 10 MHz crystal or
//ceramic resonator.
```

```
//*********************************************************************

void delay(unsigned int number_of_6_55ms_interrupts)
{
TCNT0 = 0x00;                          //reset timer0
input_delay = 0;                       //reset timer0 overflow counter
while(input_delay <= number_of_6_55ms_interrupts)
  {
  ;                                    //wait for number of interrupts
  }
}

//*********************************************************************
//initialize_ports: provides initial configuration for I/O ports
//*********************************************************************

void initialize_ports(void)
{
DDRA =0xff;  //set PORTA as output
PORTA=0x01;  //initialize low

DDRB =0x0a;  //PORTB[7:4] as input, set PORTB[3:0] as output
PORTB=0x00;  //initialize low

DDRC =0xff; //set PORTC as output
PORTC=0x00; //initialize low

DDRD =0xff; //set PORTD as output
PORTD=0x00; //initialize low
}

//*********************************************************************
```

5.5 SUMMARY

In this chapter, we provided an introduction to the interrupt features available aboard the ATmega164, and presented a proper to program the microcontroller to respond appropriately to an interrupt event. Two representative samples: an external interrupt and an internal interrupt using two different representative compilers were used to illustrate the principles presented in the chapter.

5.6 REFERENCES AND FURTHER READING

Atmel 8-bit AVR Microcontroller with 16/32/64/128K Bytes In-System Programmable Flash, ATmega164A, ATmega164PA, ATmega324A, ATmega324PA, ATmega644A, ATmega644PA, ATmega1284, ATmega1284P, 8272C-AVR-06/11, data sheet: 8272C-AVR-06/11, Atmel, San Jose, CA, 2015.

Barrett, S. F. and Pack, D. J. *Microcontrollers Fundamentals for Engineers and Scientists*. Morgan & Claypool, San Rafael, CA, 2006. DOI: 10.2200/s00025ed1v01y200605dcs001. 138

5.7 CHAPTER PROBLEMS

1. What is the purpose of an interrupt?
2. Describe the flow of events taking place inside the ATmega164 microcontroller when an interrupt occurs.
3. Describe the types of interrupts available with the ATmega164.
4. What is interrupt priority? How is it determined with the ATmega164?
5. What steps are required by the system designer to properly configure an interrupt?
6. How is the interrupt system turned "on" and "off"?
7. What is the difference between a global interrupt enable "switch" and a local interrupt enable "switch"?
8. A 10-MHz ceramic resonator is not available. Redo the example of the Timer/Counter0 Overflow interrupt provided with a time base of 1 MHz and 8 MHz.
9. What is the maximum delay that may be generated with the delay function provided in the text without any modification? How could the function be modified for longer delays?
10. Develop a 24-h timer (hh:mm:ss) using the Timer/Counter0 Overflow interrupt system. What is the accuracy of your timer? How can it be improved?

CHAPTER 6

Timing Subsystem

Objectives: After reading this chapter, the reader should be able to:

- explain key timing system related terminology;
- compute the frequency and the period of a periodic signal using a microcontroller;
- describe functional components of a typical microcontroller timer system;
- illustrate the procedure to capture incoming signal events using a microcontroller;
- illustrate the procedure to generate time critical output signals using a microcontroller;
- describe the timing related features of the Microchip ATmega164;
- describe the four operating modes of the Microchip ATmega164 timer system;
- configure the ATmega164's Timer 0, Timer 1, and Timer 2 systems; and
- program the ATmega164 timer system for a variety of applications.

6.1 OVERVIEW

One of the most important reasons for using microcontrollers in embedded systems is the capability of a microcontroller to perform time related tasks. In a simple timer application, one can program a microcontroller system to turn on or turn off an external device at a programmed time using a signal generated by a microcontroller. In a more involved application, we can use a microcontroller to generate complex digital waveforms with varying pulse widths to control the speed of a DC motor.

In this chapter we present the capabilities of the Microchip ATmega164 microcontroller family to perform time related functions. We begin with a review of timing related terminology. We then overview the timing system's general operation and features aboard the ATmega164. Next, we present a detailed discussion of each of its timing subsystems: Timer 0, Timer 1, and Timer 2, and their different modes of operation.

6.2 TIMING-RELATED TERMINOLOGY

6.2.1 FREQUENCY

Consider signal $x(t)$ that repeats itself. We call this signal periodic with period T, if it satisfies

$$x(t) = x(t + T).$$

To measure the frequency of a periodic signal, we count the number of times a particular event repeats within a one second period. The unit of frequency is the Hertz or cycles per second. For example, a sinusoidal signal with the 60 Hz frequency means that a full cycle of a sinusoid signal repeats 60 times each second or every 16.67 ms.

6.2.2 PERIOD

The reciprocal of a frequency is the period of a waveform. If an event occurs with a rate of 1 Hz, the period of that event is 1 s. To find a period, given a frequency of a signal, or vice versa, we simply need to remember their inverse relationship $f = \frac{1}{T}$, where f and T represent a frequency and the corresponding period, respectively. Both the periods and frequencies of signals are often used to specify timing constraints of embedded systems.

Example: When you are driving on a wintery road and your car is slipping, the engineers who designed your car configured the anti-slippage unit to react within some millisecond period, say 20 ms. The particular constraint would have forced the design team to program the monitoring system to check a slippage at a rate of 50 Hz.

6.2.3 DUTY CYCLE

In many time critical applications, periodic pulses are used as control signals. A good example is the use of a periodic pulse to control a servo motor. To control the direction and sometimes the speed of a motor, a periodic pulse with a changing duty cycle over time is used. The periodic pulse shown in Figure 6.1a is on (logic high) for 50% of the signal period and off (logic low) for the rest of the period. The pulse shown in (b) is on for only 25% of the same period as the signal in (a) and off for 75% of the period. The duty cycle is defined as the percentage a signal is on over one period. Therefore, we call the signal in Figure 6.1a as a periodic pulse with a 50% duty cycle and the corresponding signal in (b), a periodic pulse with a 25% duty cycle.

6.3 TIMING SYSTEM OVERVIEW

The heart of a timer system is the time base. The microcontroller's time base frequency or clock rate is used to generate a baseline clock signal that is used to update the contents of a special register called a free-running counter. The job of a free-running counter is to count up (increment) each time a rising edge (or a falling edge) is generated by a clock signal. Thus, if a clock is running at the rate of 2 MHz, the free-running counter will count up each 0.5 μs. All

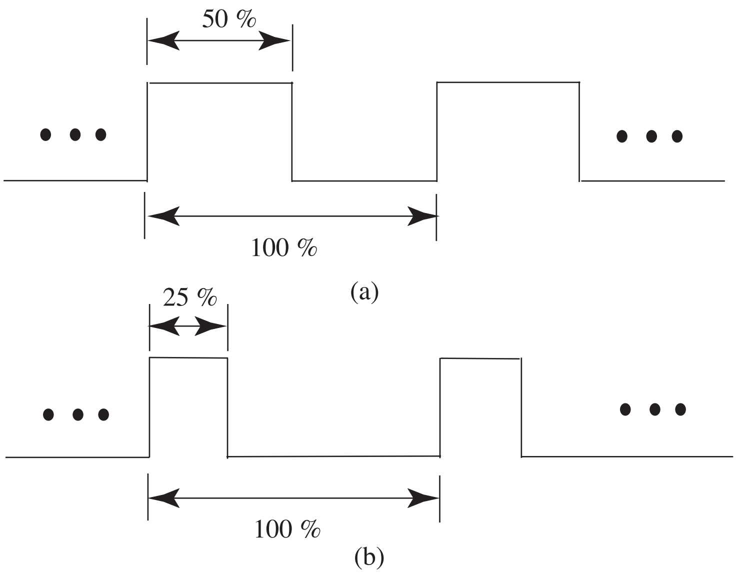


Figure 6.1: Two signals with the same period but different duty cycles. Frame (a) shows a periodic signal with a 50% duty cycle and frame (b) displays a periodic signal with a 25% duty cycle.

other time-related units reference the contents of the free-running counter to perform input and output time-related activities: measurement of time periods, capture of timing events, and generation of time-related signals.

The ATmega164 may be clocked internally using a user-selectable resistor capacitor (RC) time base or it may be clocked externally. The RC internal time base is selected using programmable fuse bits. We will discuss how to do this in the application section of this chapter. You may choose an internal fixed clock operating frequency of 128 kHz, 1 MHz, or 8 MHz.

To provide for a wider range of frequency selections, an external time source may be used. The external time sources, in order of increasing accuracy and stability, are a ceramic resonator or a crystal oscillator. The system designer chooses the time base frequency and clock source device appropriate for the application at hand. Generally speaking, a microcontroller is operated at the lowest possible frequency for a given application since clock speed is linearly related to power consumption. As previously mentioned, the maximum operating frequency of the ATmega164P with a 5 VDC supply voltage is 20 MHz.

The first primary function of a timer system is to capture the time of an external event and use it to execute appropriate instructions. To detect input time related activities, all microcontrollers typically have timer hardware components that detect signal logic changes on one or more input pins. Such components rely on a free-running counter to capture external event times. We can use this feature to measure the period, the width of a pulse, and the time of a logic change of an incoming signal.

For output timer functions, a microcontroller uses a comparator, a free-running counter, logic switch circuitry, and special purpose registers to generate time-related signals on one or more output pins. A comparator checks the value of the free-running counter for a match with the contents of another special purpose register, where a specified time in terms of the free-running counter value has been stored. The match checking process is executed at each clock cycle and when a match occurs, the corresponding hardware system induces a programmed logic change on a specific output port pin. Using such a capability, one can generate a simple logic change at a designated time, a pulse with a desired time width, or a pulse width modulated signal to control servo or direct current (DC) motors.

You can also use the timer input system to measure the pulse width of an aperiodic signal. For example, suppose that the times for the rising edge and the falling edge of an incoming signal are 1.5 s and 1.6 s, respectively. We can use free-running counter values that reflect these times to easily compute the pulse width of 0.1 s.

The second primary function of a timer system is to generate signals to control external devices. Again an output timer event simply means a change of logic states on an output pin of a microcontroller at a specified time. Now consider Figure 6.2. Suppose an external device connected to the microcontroller requires a pulse signal to turn itself on. Suppose the particular pulse the external device needs is 2 ms wide. In such a situation, we can use the free-running counter value to synchronize the time of desired logic state changes. Naturally, extending the same capability, we can also generate a periodic pulse with a fixed duty cycle or a varying duty cycle.

From the examples we discussed above, you may have wondered how a microcontroller can be used to compute absolute times from the relative free-running counter values, say 1.5 s and 1.6 s. To do so, a programmer must use the relative system clock values and derive the absolute time values.

Example: Suppose your microcontroller is clocked by a 2 MHz signal and the system clock uses a 16-bit free-running counter. For such a system, each clock period represents 0.5 μs and it takes approximately 32.78 ms to count from 0–2^{16} (65,536). The timer input system then uses the clock values to compute frequencies, periods, and pulse widths. Suppose you want to measure a pulse width of an incoming aperiodic signal. If the rising edge and the falling edge occurred at count values \$0010 and \$0114,[1] can you find the pulse width when the free-running counter is counting at 2 MHz? Let's first convert the two values into their corresponding decimal values, 16 and 276. The pulse width of the signal in the number of counter value is 260. Since we already know how long it takes for the system to count one, we can readily compute the pulse width as $260 \times 0.5\ \mu\text{s} = 130\ \mu\text{s}$.

[1] The $ symbol indicates the value that follows is in a hexadecimal form.

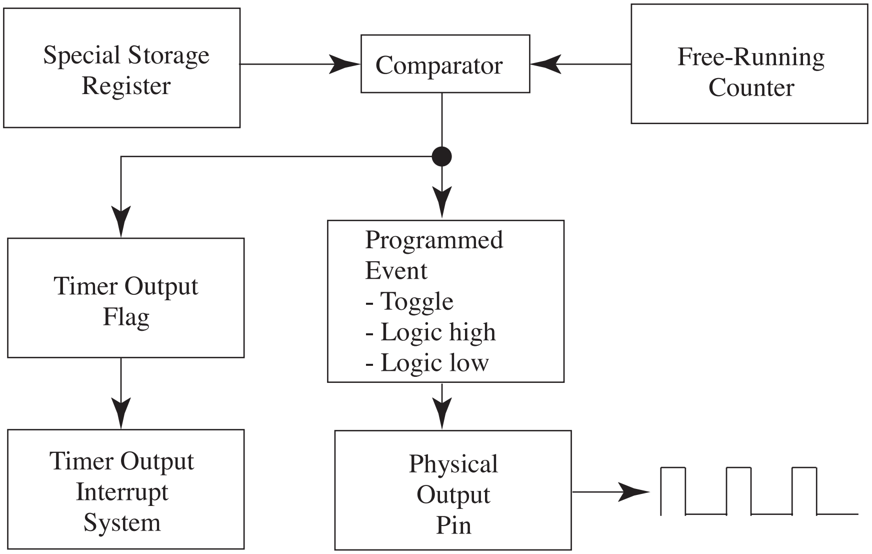


Figure 6.2: A diagram of a timer output system.

In the example above, our calculations do not take into account time increments lasting longer than the rollover time of the counter. When a counter rolls over from its maximum value back to zero, a flag is set to notify the processor of this event. The rollover events should be counted to correctly determine the overall elapsed time of an event.

Elapsed time is calculated using:

$$\begin{aligned}\text{elapsed clock ticks} &= \left(n \times 2^b\right) + (\text{stop count} - \text{start count})[\text{clock ticks}]\\ \text{elapsed time} &= (\text{elapsed clock ticks}) \times (\text{FRC clock period})[\text{seconds}].\end{aligned}$$

In this first equation, "*n*" is the number of timer overflow flag (TOF) events that occur between the start and stop events, and "*b*" is the number of bits in the timer counter. The equation yields the elapsed time in clock ticks. To convert to seconds the number of clock ticks are multiplied by the period of the clock source of the free-running counter (FRC).

6.4 APPLICATIONS

In this section, we consider some important uses of the timer system of a microcontroller: (1) to measure an input signal timing event, called input capture; (2) to count the number of external signal occurrences; (3) to generate timed signals—called output compare; and, finally; (4) to generate pulse width modulated signals. We first start with a case of measuring the time duration of an incoming signal.

6.4.1 INPUT CAPTURE—MEASURING EXTERNAL EVENT TIMING

In many applications, we are interested in measuring the elapsed time or the frequency of an external event using a microcontroller. Using the hardware and functional units discussed in the previous sections, we now present a procedure to accomplish the task of computing the frequency of an incoming periodic signal. Figure 6.3 shows an incoming periodic signal to a microcontroller.

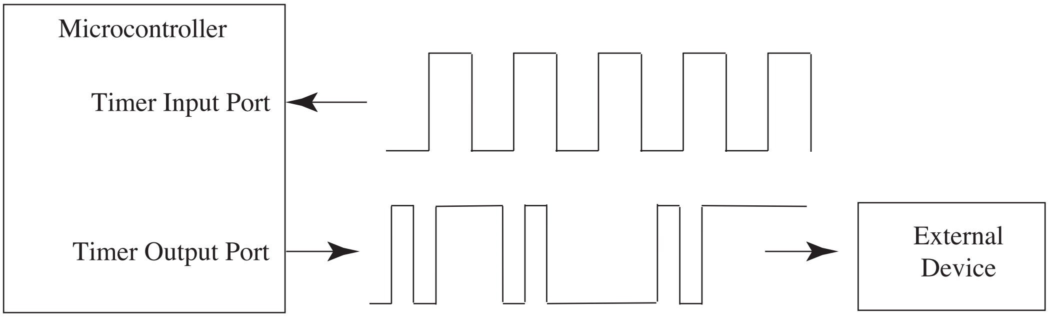


Figure 6.3: Use of the timer input and output systems of a microcontroller. The signal on top is fed into a timer input port. The captured signal is subsequently used to compute the input signal frequency. The signal on the bottom is generated using the timer output system. The signal is used to control an external device.

The first step with input capture is to turn on the timer system. To reduce power consumption, a microcontroller usually does not turn on all of its functional systems after reset until they are needed. In addition to a separate timer module, many microcontroller manufacturers allow a programmer to choose the rate of a separate timer clock that governs the overall functions of a timer module.

Once the timer is turned on and the clock rate is selected, a programmer must configure the physical port where the incoming signal arrives. This step is done using a special input timer port configuration register. The next step is to set the input system to the capture mode and specify the event of interest (e.g., rising edge, falling edge). In our current example, we should capture two consecutive rising edges or falling edges of the incoming signal. Again, the configuration is done by storing appropriate bits to timer related registers.

Now that the input timer system is configured appropriately, you now have two options to accomplish the input capture task. The first one is the use of a polling technique. In this method, the microcontroller continuously polls a flag, which holds a logic high signal when a programmed event occurs on the physical pin. Once the microcontroller detects the flag, the flag needs to be cleared and the time from the free-running counter when the flag occurred needs to be recorded in a special register.

The program needs to continue to wait for the next flag which indicates the end of one period of the incoming signal. The programmer then needs to record the newly acquired captured time represented in the form of a free-running counter value again.

The period of the signal can now be computed by computing the time difference between the two captured event times, and, based on the clock speed of the microcontroller, the programmer can compute the actual time changes and consequently the frequency of the signal.

In many cases, a microcontroller can't afford the time to poll for a single event. This brings us to the second method: interrupt systems. As discussed in Chapter 5, most microcontrollers are equipped with built-in interrupt systems with timer input modules. Instead of continuously polling for a flag, a microcontroller may perform other tasks and relies on its interrupt system to detect the programmed event.

The task of computing the period and the frequency is the same as the first method, except that the microcontroller will not be tied down to constantly checking the flag, increasing the efficient use of the microcontroller resources. To use interrupt systems, of course, we must appropriately configure the appropriate interrupt system to be triggered when a desired event is detected and develop an associated interrupt service routine.

Example: Suppose that for an input capture scenario, the two captured times for the two rising edges are \$1000 and \$5000, respectively. Note that these values are not absolute times but the representations of times reflected as the values of the free-running counter. The period of the signal is \$4000 or 16384 in a decimal form. If we assume that the timer clock runs at 10 MHz, the period of the signal is 1.6384 ms, and the corresponding frequency of the signal is approximately 610.35 Hz.

6.4.2 COUNTING EVENTS

The same capability of measuring the period of a signal can also be used to simply count external events. Suppose we want to count the number of logic state changes of an incoming signal for a given period of time. Again, we can use the polling or the interrupt technique to accomplish the task. For both techniques, the initial steps of turning on a timer and configuring a physical input port pin are the same. In this application, however, the programmed event should be any logic state changes instead of looking for a rising or a falling edge as we have done in the previous section. If the polling technique is used, at each event detection, the corresponding flag must be cleared and a counter must be updated. If the interrupt technique is used, one must write an interrupt service routine within which the flag is cleared and a counter is updated.

6.4.3 OUTPUT COMPARE—GENERATING TIMING SIGNALS TO INTERFACE EXTERNAL DEVICES

In the previous two sections, we considered two applications of capturing external incoming signals. In this section and the next one, we consider how a microcontroller can generate time critical signals for external devices. Suppose that in this application we want to send a signal shown in Figure 6.3 to turn on an external device. The timing signal we have chosen is arbitrary but the application will show that a timer output system can generate any desired time-related signals permitted under the timer clock speed limit of a microcontroller.

Similar to the use of the timer input system, one must first turn on the timer system and configure a physical pin as a timer output pin using designated registers. In addition, one also needs to program the desired external event using another special register associated with the timer output system. To generate the signal shown in Figure 6.3, one must compute the time required between the rising and the falling edges. Suppose that the external device requires a pulse which is 2 ms wide to be activated. To generate the desired pulse, one must first program the logic state for the particular pin to be low and set the time value using a special register with respect to the contents of the free-running counter. At each clock cycle, the special register contents are compared with the contents of the free-running counter and when a match occurs, the programmed logic state appears on the designated hardware pin. Once the rising edge is generated, the program then must reconfigure the event to be a falling edge (logic state low) and change the contents of the special register to be compared with the free-running counter.

Example: For the particular example in Figure 6.3, let's assume that the main clock runs at 2 MHz, the free-running counter is a 16-bit counter, and the name of the special register (16-bit register) where we can put appropriate values is output timer register. To generate the desired pulse, we can first put $0000 to the output timer register, and after the rising edge has been generated, we need to change the program event to a falling edge and put $0FA0 or 4000 in decimal to the output timer register. As was the case with the input timer system module, we can use output timer system interrupts to generate the desired signals as well.

6.4.4 PULSE WIDTH MODULATION (PWM)

In this section, we discuss a well-known method to control the speed of a DC motor using a pulse width modulated (PWM) signal. The underlying concept is as follows. If we turn on a DC motor and provide it with the required voltage, the motor will run at its maximum speed. Now, suppose we turn the motor on and off rapidly, by applying a periodic signal. The motor at some point cannot react fast enough to the changes of the voltage values and will run at the speed proportional to the average time the motor was turned on. Using this principle and by changing the duty cycle, we can control the speed of a DC motor as we desire. Suppose again we want to generate a speed profile shown in Figure 6.4. As shown in the figure, we want to accelerate the speed, maintain the speed, and decelerate the speed of the motor for a fixed amount of time.

The first necessary task is again to turn on the timer system, configure a physical port, and program the event to be a rising edge. As a part of the initialization process, we need to put $0000 to the output timer register we discussed in the previous section. Once the rising edge is generated, the program then needs to modify the event to a falling edge and change the contents of the output timer register to a value proportional to a desired duty cycle. For example, if we

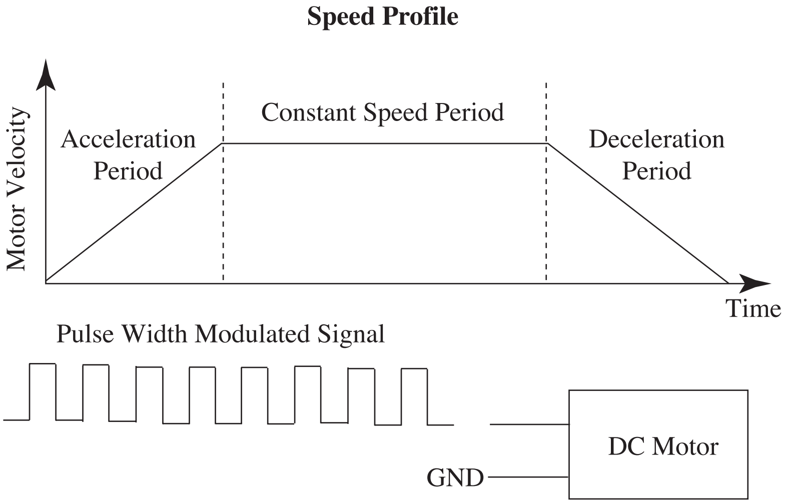


Figure 6.4: The figure shows the speed profile of a DC motor over time when a pulse-width-modulated signal is applied to the motor.

want to start off with 25% duty cycle, we need to input $4000 to the register, provided that we are using a 16-bit free-running counter. Once the falling edge is generated, we now need to go back and change the event to be a rising edge and the contents of the output timer counter value back to $0000. If we want to continue to generate a 25% duty cycle signal, then we must repeat the process indefinitely. Note that we are using the time for a free-running counter to count from $0000 to $FFFF as one period.

Now suppose we want to increase the duty cycle to 50% over 1 s and that the clock is running at 2 MHz. This means that the free-running counter counts from $0000–$FFFF every 32.768 ms, and the free-running counter will count from $0000–$FFFF approximately 30.51 times over the period of 1 s. That is we need to incrementally increase the pulse width from $4000 to $8000 in approximately 30 iterations, or approximately 546 clock counts every iteration.

6.5 OVERVIEW OF THE MICROCHIP TIMERS

The Microchip ATmega164 is equipped with a flexible and powerful three channel timer system. The three timer channels are designated Timer 0, Timer 1, and Timer 2. In this section we review the operation of the timer system in detail. We begin with an overview of the timer system features followed by a detailed discussion of Timer Channel 0. Space does not permit a complete discussion of the other two timing channels; however, we review their complement of registers and highlight their features not contained in our discussion of Timer Channel 0. The information provided on Timer Channel 0 is readily adapted to the other two channels.

The features of the timing system are summarized in Figure 6.5. Timer 0 and Timer 2 are 8-bit timers; whereas, Timer 1 is a 16-bit timer. Each timer channel is equipped with a prescaler. The prescaler is used to subdivide the main microcontroller clock source (designated $f_{clk\_I/O}$ in upcoming diagrams) down to the clock source for the timing system (clk_{Tn}).

Timer 0	**Timer 1**	**Timer 2**
- 8-bit timer/counter - 10-bit clock prescaler - Functions: -- Pulse width modulation -- Frequency generation -- Event counter -- Output compare -- 2 ch - Modes of operation: -- Normal -- Clear timer on compare match (CTC) -- Fast PWM -- Phase correct PWM	- 16-bit timer/counter - 10-bit clock prescaler - Functions: -- Pulse width modulation -- Frequency generation -- Event counter -- Output compare -- 2 ch -- Input capture - Modes of operation: -- Normal -- Clear timer on compare match (CTC) -- Fast PWM -- Phase correct PWM	- 8-bit timer/counter - 10-bit clock prescaler - Functions: -- Pulse width modulation -- Frequency generation -- Event counter -- Output compare -- 2 ch - Modes of operation: -- Normal -- Clear timer on compare match (CTC) -- Fast PWM -- Phase correct PWM

Figure 6.5: Microchip timer system overview.

Each timer channel has the capability to generate pulse width modulated signals, generate a periodic signal with a specific frequency, count events, and generate a precision signal using the output compare functions. Additionally, Timer 1 is equipped with the Input Capture feature.

All timer channels may be configured to operate in one of four operational modes designated: Normal, Clear Timer on Compare Match (CTC), Fast PWM, and Phase Correct PWM. We provide more information on these modes shortly.

6.6 TIMER 0 SYSTEM

In this section we discuss the features, overall architecture, modes of operation, registers, and programming of Timer 0. This information may be readily adapted to Timer 1 and Timer 2.

A Timer 0 block diagram is shown in Figure 6.6. The clock source for Timer 0 is provided via an external clock source at the T0 pin (PB0) of the microcontroller. Timer 0 may also be clocked internally via the microcontroller's main clock ($f_{clk\_I/O}$). This clock frequency may be too rapid for many applications. Therefore, the timing system is equipped with a prescaler to

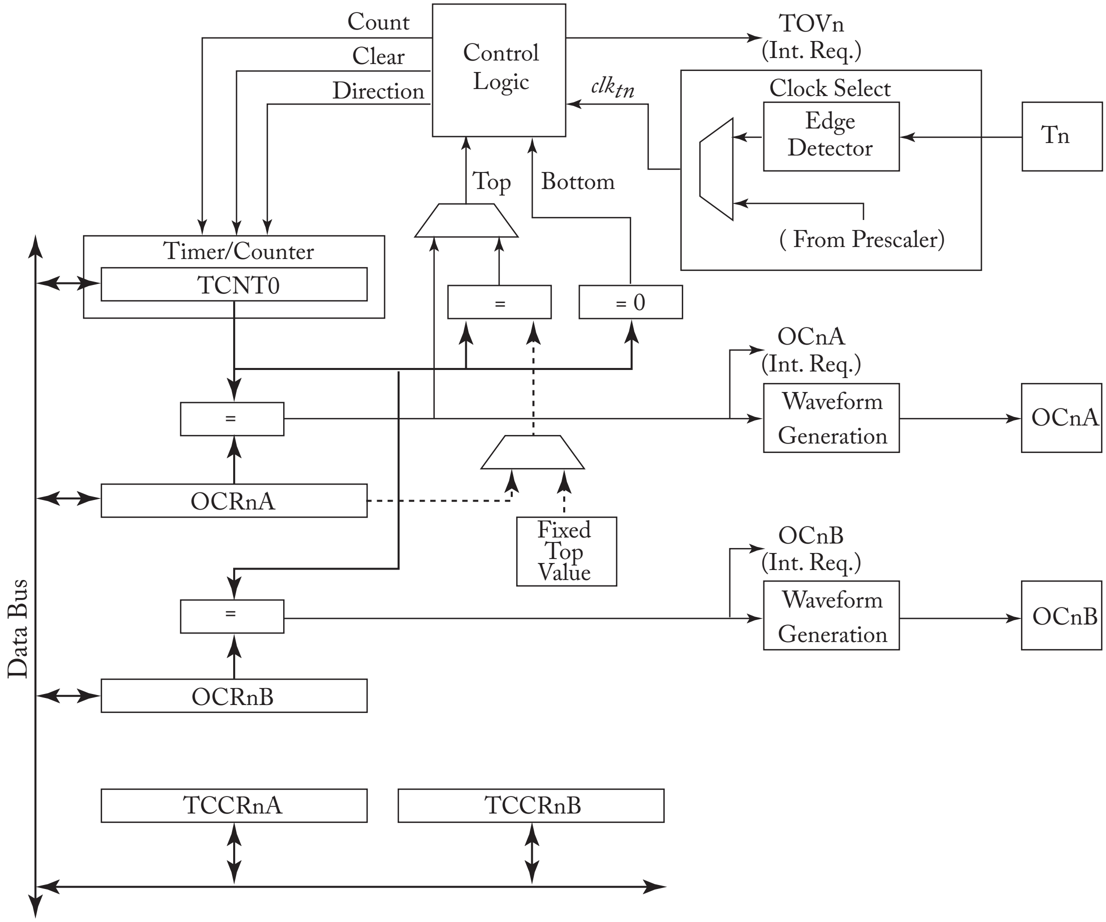


Figure 6.6: Timer 0 block diagram. Figure used with permission Microchip Technology, Inc. (www.microchip.com).

subdivide the main clock frequency down to timer system frequency (clk_{Tn}). The clock source for Timer 0 is selected using the CS0[2:0] bits contained in the Timer/Counter Control Register B (TCCR0B). The TCCR0A register contains the WGM0[1:0] bits and the COM0A[1:0] (and B) bits. Whereas, the TCCR0B register contains the WGM0[2] bit. These bits are used to select the mode of operation for Timer 0 as well as tailor waveform generation for a specific application.

The timer clock source (clk_{Tn}) is fed to the 8-bit Timer/Counter Register (TCNT0). This register is incremented (or decremented) on each clk_{Tn} clock pulse. Timer 0 is also equipped with two 8-bit comparators that constantly compares the numerical content of TCNT0 to the output compare register A (OCR0A) and output compare register B (OCR0B). The compare signal from the 8-bit comparator is fed to the waveform generators. The waveform generators have a number of inputs to perform different operations with the timer system.

The BOTTOM signal for the waveform generation and the control logic, shown in Figure 6.7, is asserted when the timer counter TCNT0 reaches all zeroes (0x00). The MAX signal for the control logic unit is asserted when the counter reaches all ones (0xFF). The TOP signal for the waveform generation is asserted by either reaching the maximum count values of 0xFF on the TCNT0 register or reaching the value set in the output compare register 0 A (OCR0A) or B. The setting for the TOP signal will be determined by the Timer's mode of operation.

Timer 0 also uses certain bits within the timer/counter interrupt mask register 0 (TIMSK0) and the timer/counter interrupt flag register 0 (TIFR0) to configure interrupt related events.

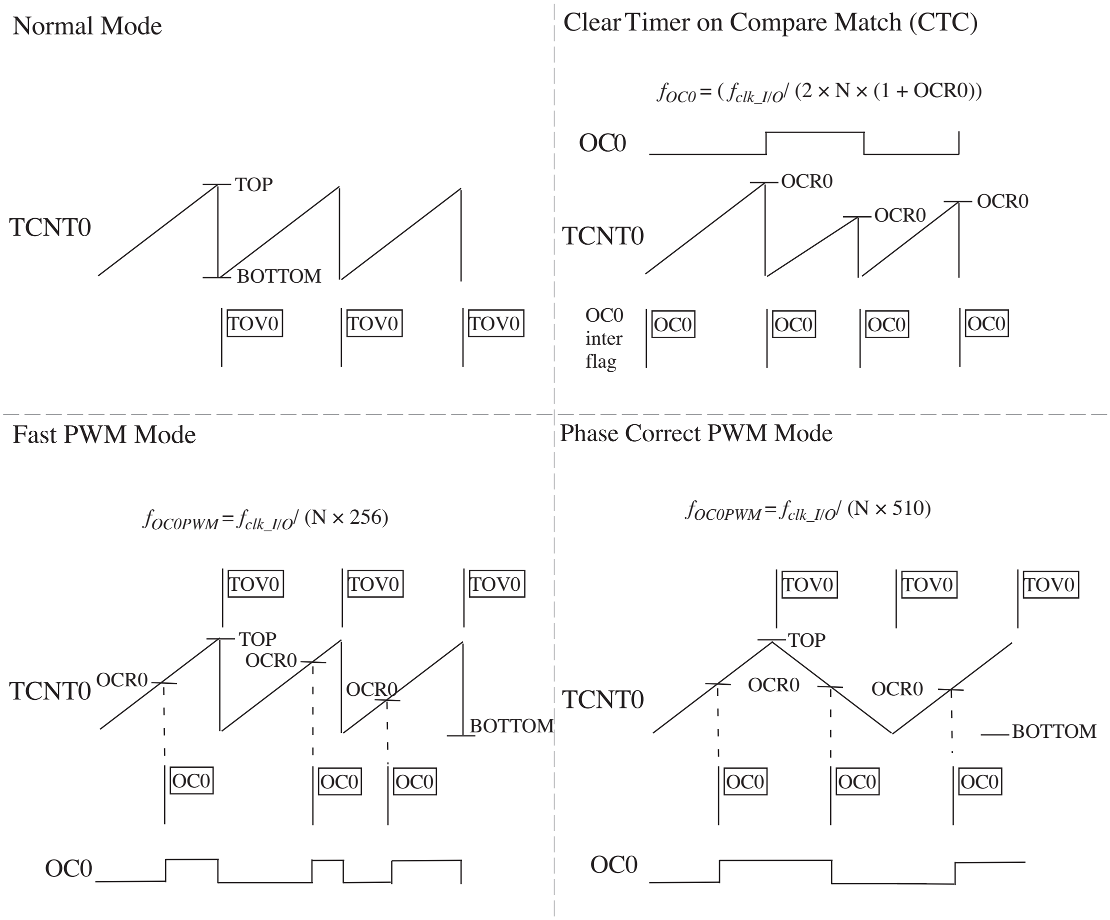


Figure 6.7: Timer 0 modes of operation.

6.6.1 MODES OF OPERATION

Each of the timer channels may be set for a specific mode of operation: normal, clear timer on compare match (CTC), fast PWM, and phase correct PWM. The system designer chooses the correct mode for the application at hand. The timer modes of operation are summarized in Figure 6.7. A specific mode of operation is selected using the Waveform Generation Mode bits located in timer/control register A (TCCR0A) and timer/control register B (TCCR0B).

Normal Mode

In the normal mode the timer will continually count up from 0x00 (BOTTOM) to 0xFF (TOP). When the TCNT0 returns to zero on each cycle of the counter the timer/counter overflow flag (TOV0) will be set. The normal mode is useful for generating a periodic "clock tick" that may be used to calculate elapsed real time or to provide delays within a system.

Clear Timer on Compare Match (CTC)

In the CTC mode, the TCNT0 timer is reset to zero every time the TCNT0 counter reaches the value set in output compare register A (OCR0A) or B. The output compare flag A (OCF0A) or B is set when this event occurs. The OCF0A or B flag is enabled by asserting the Timer/Counter 0 output compare math interrupt enable (OCIE0A) or B flag in the Timer/Counter interrupt mask register 0 (TIMSK0) and when the I-bit in the Status Register is set to one.

The CTC mode is used to generate a precision digital waveform such as a periodic signal or a single pulse. The user must describe the parameters and key features of the waveform in terms of Timer 0 "clock ticks." When a specific key feature is reached within the waveform the next key feature may be set into the OCR0A or B register.

Phase Correct PWM Mode

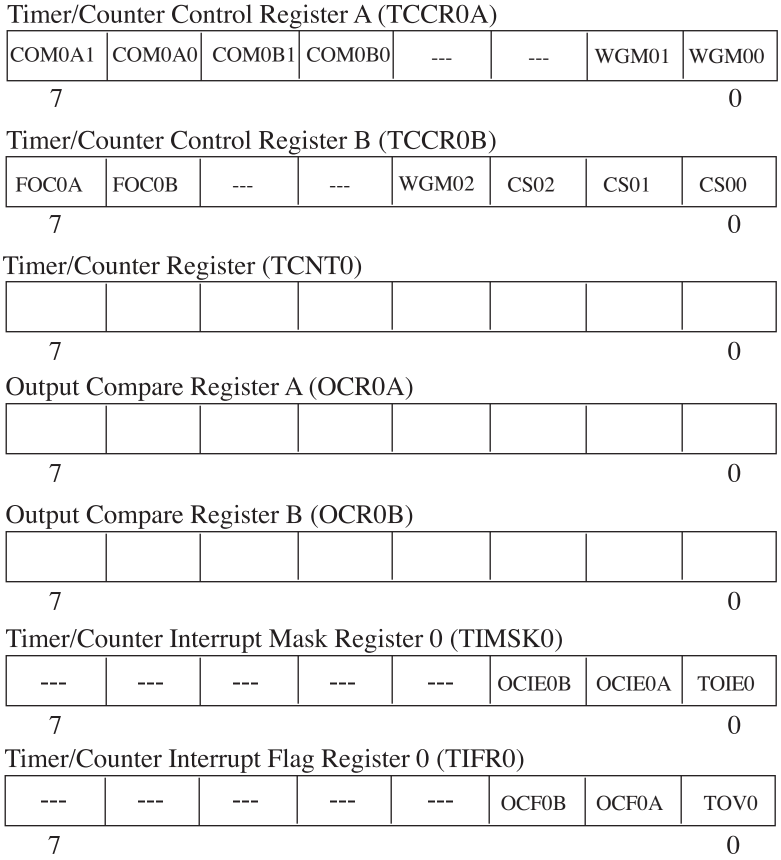
In the Phase Correct PWM Mode, the TCNT0 register counts from 0x00 to 0xFF and back down to 0x00 continually. Every time the TCNT0 value matches the value set in the OCR0A or B register, the OCF0A or B flag is set and a change in the PWM signal occurs.

Fast PWM

The Fast PWM mode is used to generate a precision PWM signal of a desired frequency and duty cycle. It is called the Fast PWM because its maximum frequency is twice that of the Phase Correct PWM mode. When the TCNT0 register value reaches the value set in the OCR0A or B register, it will cause a change in the PWM output as prescribed by the system designer. It continues to count up to the TOP value at which time the Timer/Counter 0 Overflow Flag is set.

6.6.2 TIMER 0 REGISTERS

A summary of the Timer 0 registers is shown in Figure 6.8.



Timer/Counter Control Register A (TCCR0A)

COM0A1	COM0A0	COM0B1	COM0B0	---	---	WGM01	WGM00
7							0

Timer/Counter Control Register B (TCCR0B)

FOC0A	FOC0B	---	---	WGM02	CS02	CS01	CS00
7							0

Timer/Counter Register (TCNT0)

7							0

Output Compare Register A (OCR0A)

7							0

Output Compare Register B (OCR0B)

7							0

Timer/Counter Interrupt Mask Register 0 (TIMSK0)

---	---	---	---	---	OCIE0B	OCIE0A	TOIE0
7							0

Timer/Counter Interrupt Flag Register 0 (TIFR0)

---	---	---	---	---	OCF0B	OCF0A	TOV0
7							0

Figure 6.8: Timer 0 registers.

Timer/Counter Control Registers A and B (TCCR0A and TCCR0B)

The TCCR0 register bits are used to:

- select the operational mode of Timer 0 using the Waveform Mode Generation (WGM0[2:0]) bits,
- determine the operation of the timer within a specific mode with the Compare Match Output Mode (COM0A[1:0] or COM0B[1:0] or) bits, and
- select the source of the Timer 0 clock using Clock Select (CS0[2:0]) bits.

The bit settings for the TCCR0 register are summarized in Figure 6.9.

CS0[2:0]	Clock Source
000	None
001	$clk_{I/0}$
010	$clk_{I/0}/8$
011	$clk_{I/0}/64$
100	$clk_{I/0}/8clk_{I/0}/256$
101	$clk_{I/0}/8clk_{I/0}/1024$
110	External clock on T0 (falling edge trigger)
111	External clock on T1 (rising edge trigger)

Clock Select

Timer/Counter Control Register B (TCCR0B)

FOC0A	FOC0B	---	---	WGM02	CS02	CS01	CS00

70

Timer/Counter Control Register A (TCCR0A)

COM0A1	COM0A0	COM0B1	COM0B0	---	---	WGM01	WGM00

Waveform Generation Mode

Mode	WGM[02:00]	Mode
0	000	Normal
1	001	PWM, Phase Correct
2	010	CTC
3	011	Fast PWM
4	100	Reserved
5	101	PWM, Phase Correct
6	110	Reserved
7	111	Fast PWM

Compare Output Mode, non-PWM Mode

COM0A[1:0]	Description
00	Normal, OC0A disconnected
01	Toggle OC0A on compare match
10	Clear OC0A on compare match
11	Set OC0A on compare match

Compare Output Mode, non-PWM Mode

COM0B[1:0]	Description
00	Normal, OC0B disconnected
01	Toggle OC0B on compare match
10	Clear OC0B on compare match
11	Set OC0B on compare match

Compare Output Mode, Fast PWM Mode

COM0A[1:0]	Description
00	Normal, OC0A disconnected

01	WGM02 = 0: normal operation, OC0A disconnected
	WGM02 = 1:Toggle OC0A on compare match

10	Clear OC0A on compare match, set OC0A at Bottom (non-inverting mode)
11	Set OC0A on compare match, clear OC0A at Bottom (inverting mode)

Compare Output Mode, Fast PWM Mode

COM0B[1:0]	Description
00	Normal, OC0B disconnected
01	Reserved
10	Clear OC0B on compare match, set OC0B at Bottom (non-inverting mode)
11	Set OC0B on compare match, clear OC0B at Bottom (inverting mode)

Compare Output Mode, Phase Correct PWM

COM0A[1:0]	Description
00	Normal, OC0A disconnected

01	WGM02 = 0: normal operation, OC0A disconnected
	WGM02 = 1:Toggle OC0A on compare match

10	Clear OC0A on compare match, when upcounting. Set OC0A on compare match when down counting
11	Set OC0A on compare match, when upcounting. Set OC0A on compare match when down counting

Compare Output Mode, Phase Correct PWM

COM0B[1:0]	Description
00	Normal, OC0B disconnected
01	Reserved
10	Clear OC0B on compare match, when upcounting. Set OC0B on compare match when down counting
11	Set OC0B on compare match, when upcounting. Set OC0B on compare match when down counting

Figure 6.9: Timer/counter control registers A and B (TCCR0A and TCCR0B) bit settings.

Timer/Counter Register(TCNT0)

The TCNT0 is the 8-bit counter for Timer 0.

Output Compare Registers A and B (OCR0A and OCR0B)

The OCR0A and B registers each hold a user-defined 8-bit value that is continuously compared to the TCNT0 register.

Timer/Counter Interrupt Mask Register (TIMSK0)

Timer 0 uses the Timer/Counter 0 Output Compare Match Interrupt Enable A and B (OCIE0A and B) bits and the Timer/Counter 0 Overflow Interrupt Enable (TOIE0) bit. When the OCIE0A or B bit and the I-bit in the Status Register are both set to one, the Timer/Counter 0 Compare Match interrupt is enabled. When the TOIE0 bit and the I-bit in the Status Register are both set to one, the Timer/Counter 0 Overflow interrupt is enabled.

Timer/Counter Interrupt Flag Register 0 (TIFR0)

Timer 0 uses the Output Compare Flag A or B (OCF0A and OCF0B) which sets for an output compare match. Timer 0 also uses the Timer/Counter 0 Overflow Flag (TOV0) which sets when the contents of the Timer/Counter 0 overflows.

6.7 TIMER 1

Timer 1 uses a 16-bit timer/counter. It shares many of the same features of the Timer 0 channel. Due to limited space the shared information will not be repeated. Instead we concentrate on the enhancements of Timer 1 which include an additional output compare channel and also the capability for input capture. The block diagram for Timer 1 is shown in Figure 6.10.

As discussed earlier in the chapter, the input capture feature is used to capture the characteristics of an input signal including period, frequency, duty cycle, or pulse length. This is accomplished by monitoring for a user-specified edge on the ICP1 microcontroller pin. When the desired edge occurs, the value of the timer/counter 1 (TCNT1) register is captured and stored in the input capture register 1 (ICR1).

6.7.1 TIMER 1 REGISTERS

The complement of registers supporting Timer 1 are shown in Figure 6.11. Each register will be discussed in turn.

TCCR1A and TCCR1B registers

The TCCR1 register bits are used to:

- select the operational mode of Timer 1 using the Waveform Mode Generation (WGM1[3:0]) bits,

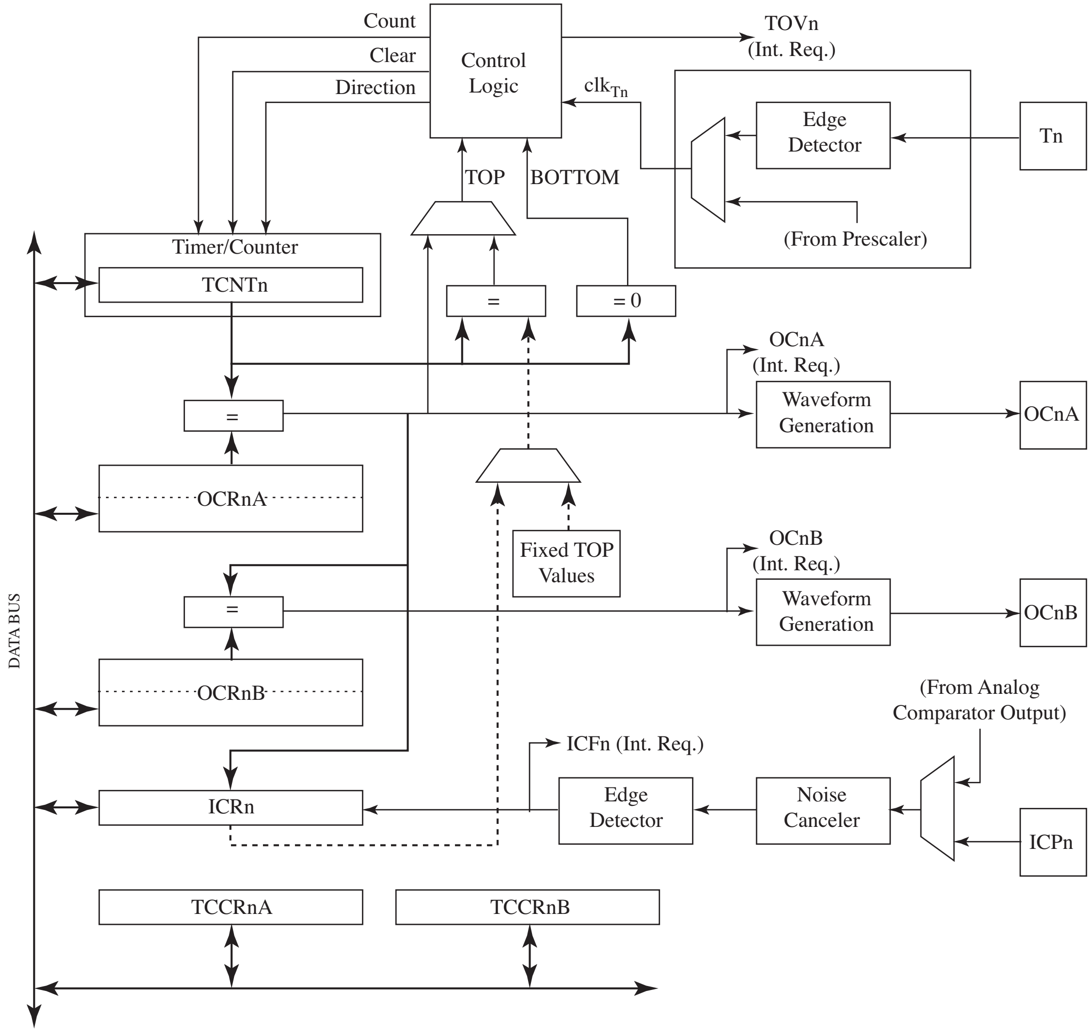


Figure 6.10: Timer 1 block diagram. (Figure used with permission Microchip Technology, Inc. (www.microchip.com).)

- determine the operation of the timer within a specific mode with the Compare Match Output Mode (Channel A: COM1A[1:0] and Channel B: COM1B[1:0]) bits, and
- select the source of the Timer 1 clock using clock select (CS1[2:0]) bits.

The bit settings for the TCCR1A and TCCR1B registers are summarized in Figure 6.12.

Timer/Counter 1 Control Register A (TCCR1A)

COM1A1	COM1A0	COM1B1	COM1B0	---	---	WGM11	WGM10
7							0

Timer/Counter 1 Control Register B (TCCR1B)

ICNC1	ICES1	---	WGM13	WGM12	CS12	CS11	CS10
7							0

Timer/Counter 1 Control Register C (TCCR1C)

FOC1A	FOC1B	---	---	---	---	---	---
7							0

Timer/Counter 1 (TCCT1H/TCNT1L)

15							8
7							0

Output Compare Register 1 A (OCR1AH/OCR1AL)

15							8
7							0

Output Compare Register 1 B (OCR1BH/OCR1BL)

15							8
7							0

Input Capture Register 1 (ICR1H/ICR1L)

15							8
7							0

Timer/Counter Interrupt Mask Register 1 (TIMSK1)

---	---	ICIE1	---	---	OCIE1B	OCIE1A	TOIE1
7							0

Timer/Counter 1 Interrupt Flag Register (TIFR1)

---	---	ICF1	---	---	OCF1B	OCF1A	TOV1
7							0

Figure 6.11: Timer 1 registers.

CS0[2:0]	Clock Source
000	None
001	$\text{clk}_{I/0}$
010	$\text{clk}_{I/0}/8$
011	$\text{clk}_{I/0}/64$
100	$\text{clk}_{I/0}/8\text{clk}_{I/0}/256$
101	$\text{clk}_{I/0}/8\text{clk}_{I/0}/1024$
110	External clock on T0 (falling edge trigger)
111	External clock on T1 (rising edge trigger)

Clock Select

Timer/Counter 1 Control Register B (TCCR1B)

ICNC1	ICES1	—	WGM13	WGM12	CS12	CS11	CS10
7							0

Timer/Counter 1 Control Register A (TCCR1A)

COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
7							0

Waveform Generation Mode

Mode	WGM[13:12:11:10]	Mode
0	0000	Normal
1	0001	PWM, Phase Correct, 8-bit
2	0010	PWM, Phase Correct, 9-bit
3	0011	PWM, Phase Correct, 10-bit
4	0100	CTC
5	0101	Fast PWM, 8-bit
6	0110	Fast PWM, 9-bit
7	0111	Fast PWM, 10-bit
8	1000	PWM, Phase & Freq Correct
9	1001	PWM, Phase & Freq Correct
10	1010	PWM, Phase Correct
11	1011	PWM, Phase Correct
12	1100	CTC
13	1101	Reserved
14	1110	Fast PWM
15	1111	Fast PWM

Normal, CTC

COMx[1:0]	Description
00	Normal, OC1A/1B disconnected
01	Toggle OC1A/1B on compare match
10	Clear OC1A/1B on compare match
11	Set OC1A/1B on compare match

PWM, Phase Correct, Phase & Freq Correct

COMx[1:0]	Description
00	Normal, OC0 disconnected
01	WGM1[3:0] = 9 or 14: toggle OCnA on compare match, OCnB disconnected WGM1[3:0]= other settings, OC1A/1B disconnected
10	Clear OC0 on compare match when up-counting. Set OC0 on compare match when down counting
11	Set OC0 on compare match when up-counting. Clear OC0 on compare match when down counting.

Fast PWM

COMx[1:0]	Description
00	Normal, OC1A/1B disconnected
01	WGM1[3:0] = 15, toggle OC1A on compare match OC1B disconnected WGM1[3:0] = other settings, OC1A/1B disconnected
10	Clear OC1A/1B on compare match, set OC1A/1B at TOP
11	Set OC1A/1B on compare match, clear OC1A/1B at TOP

Figure 6.12: TCCR1A and TCCR1B registers.

Timer/Counter Register 1 (TCNT1H/TCNT1L)

The TCNT1 is the 16-bit counter for Timer 1.

Output Compare Register 1 (OCR1AH/OCR1AL)

The OCR1A register holds a user-defined 16-bit value that is continuously compared to the TCNT1 register when Channel A is used.

OCR1BH/OCR1BL

The OCR1B register holds a user-defined 16-bit value that is continuously compared to the TCNT1 register when Channel B is used.

Input Capture Register 1 (ICR1H/ICR1L)

ICR1 is a 16-bit register used to capture the value of the TCNT1 register when a desired edge on ICP1 pin has occurred.

Timer/Counter Interrupt Mask Register 1 (TIMSK1)

Timer 1 uses the Timer/Counter 1 Output Compare Match Interrupt Enable (OCIE1A/1B) bits, the Timer/Counter 1 Overflow Interrupt Enable (TOIE1) bit, and the Timer/Counter 1 Input Capture Interrupt Enable (IC1E1) bit. When the OCIE1A/B bit and the I-bit in the Status Register are both set to one, the Timer/Counter 1 Compare Match interrupt is enabled. When the OIE1 bit and the I-bit in the Status Register are both set to one, the Timer/Counter 1 Overflow interrupt is enabled. When the IC1E1 bit and the I-bit in the Status Register are both set to one, the Timer/Counter 1 Input Capture interrupt is enabled.

Timer/Counter Interrupt Flag Register (TIFR1)

Timer 1 uses the Output Compare Flag 1 A/B (OCF1A/B) which sets for an output compare A/B match. Timer 1 also uses the Timer/Counter 1 Overflow Flag (TOV1) which sets when Timer/Counter 1 overflows. Timer Channel 1 also uses the Timer/Counter 1 Input Capture Flag (ICF1) which sets for an input capture event.

6.8 TIMER 2

Timer 2 is another 8-bit timer channel similar to Timer 0. The Timer 2 channel block diagram is provided in Figure 6.13. Its registers are summarized in Figure 6.14.

Timer/Counter Control Register A and B (TCCR2A and B)

The TCCR2A and B register bits are used to:

- select the operational mode of Timer 2 using the Waveform Mode Generation (WGM2[2:0]) bits,

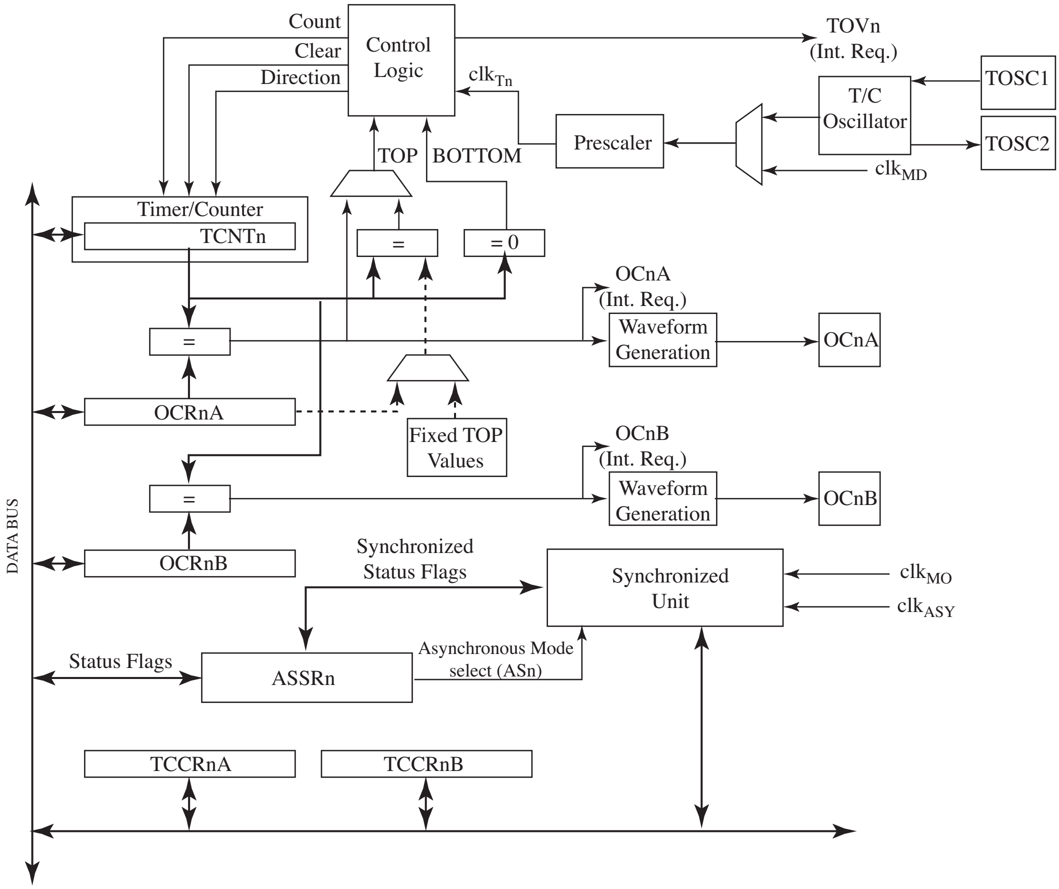


Figure 6.13: Timer 2 block diagram. (Figure used with permission Microchip Technology, Inc. (`www.microchip.com`).)

- determine the operation of the timer within a specific mode with the Compare Match Output Mode (COM2A[1:0] and B) bits, and
- select the source of the Timer 2 clock using clock select (CS2[2:0]) bits.

The bit settings for the TCCR2A and B registers are summarized in Figure 6.15.

Timer/Counter Register (TCNT2)

The TCNT2 is the 8-bit counter for Timer 2.

Timer/Counter Control Register A (TCCR2A)

COM2A1	COM2A0	COM2B1	COM2B0	---	---	WGM21	WGM20
7							0

Timer/Counter Control Register B (TCCR2B)

FOC2A	FOC2B	---	---	WGM22	CS22	CS21	CS20
7							0

Timer/Counter Register (TCNT2)

7							0

Output Compare Register A (OCR2A)

7							0

Output Compare Register B (OCR2B)

7							0

Timer/Counter 2 Interrupt Mask Register (TIMSK2)

---	---	---	---	---	OCIE2B	OCIE2A	TOIE2
7							0

Timer/Counter 2 Interrupt Flag Register (TI FR2)

---	---	---	---	---	OCF2B	OCF2A	TOV2
7							0

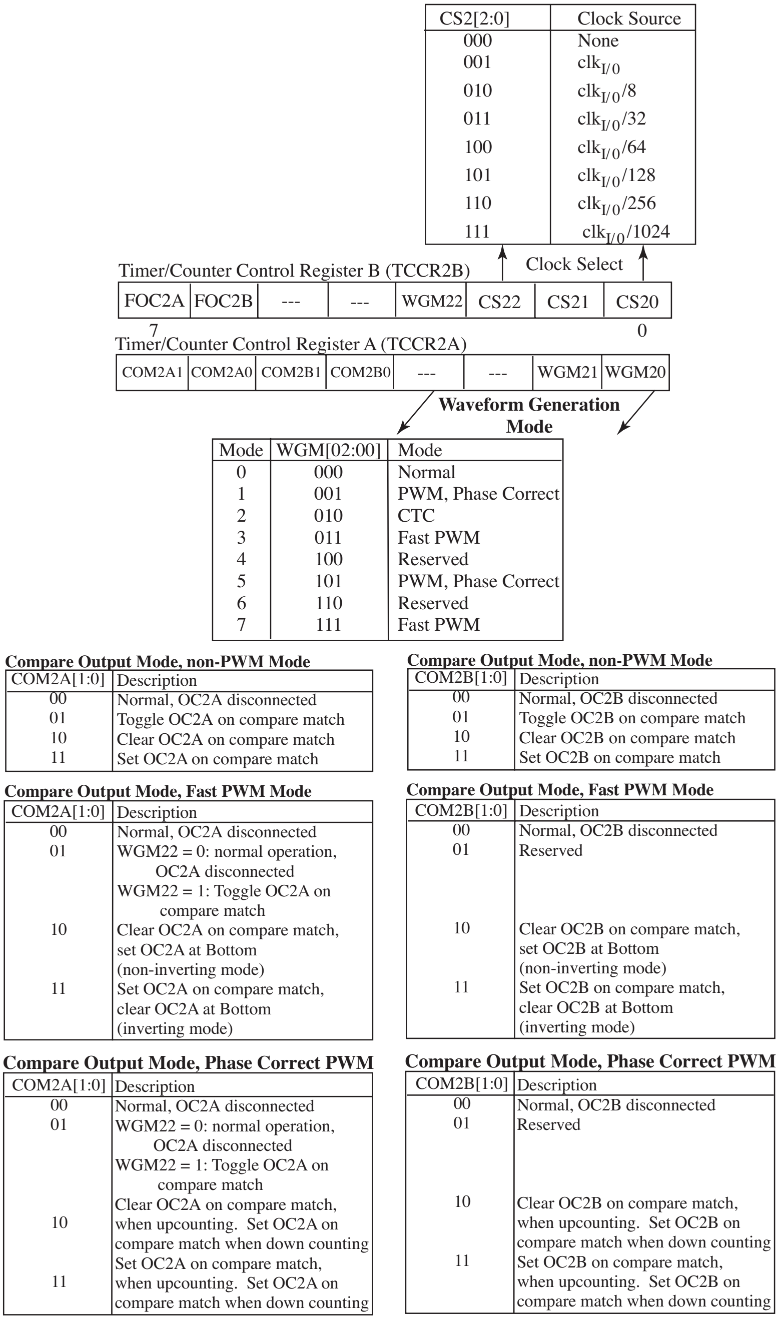
Figure 6.14: Timer 2 registers.

Output Compare Register A and B (OCR2A and B)

The OCR2A and B registers each holds a user-defined 8-bit value that is continuously compared to the TCNT2 register.

Timer/Counter Interrupt Mask Register 2 (TIMSK2)

Timer 2 uses the Timer/Counter 2 Output Compare Match Interrupt Enable A and B (OCIE2A and B) bits and the Timer/Counter 2 Overflow Interrupt Enable A and B (OIE2A and B) bits. When the OCIE2A or B bit and the I-bit in the Status Register are both set to one, the Timer/Counter 2 Compare Match interrupt is enabled. When the TOIE2 bit and the I-bit in the Status Register are both set to one, the Timer/Counter 2 Overflow interrupt is enabled.



CS2[2:0]	Clock Source
000	None
001	$clk_{I/0}$
010	$clk_{I/0}/8$
011	$clk_{I/0}/32$
100	$clk_{I/0}/64$
101	$clk_{I/0}/128$
110	$clk_{I/0}/256$
111	$clk_{I/0}/1024$

Timer/Counter Control Register B (TCCR2B)

7							0
FOC2A	FOC2B	---	---	WGM22	CS22	CS21	CS20

Timer/Counter Control Register A (TCCR2A)

COM2A1	COM2A0	COM2B1	COM2B0	---	---	WGM21	WGM20

Mode	WGM[02:00]	Mode
0	000	Normal
1	001	PWM, Phase Correct
2	010	CTC
3	011	Fast PWM
4	100	Reserved
5	101	PWM, Phase Correct
6	110	Reserved
7	111	Fast PWM

Compare Output Mode, non-PWM Mode

COM2A[1:0]	Description
00	Normal, OC2A disconnected
01	Toggle OC2A on compare match
10	Clear OC2A on compare match
11	Set OC2A on compare match

Compare Output Mode, non-PWM Mode

COM2B[1:0]	Description
00	Normal, OC2B disconnected
01	Toggle OC2B on compare match
10	Clear OC2B on compare match
11	Set OC2B on compare match

Compare Output Mode, Fast PWM Mode

COM2A[1:0]	Description
00	Normal, OC2A disconnected

01	WGM22 = 0: normal operation, OC2A disconnected
	WGM22 = 1: Toggle OC2A on compare match

10	Clear OC2A on compare match, set OC2A at Bottom (non-inverting mode)
11	Set OC2A on compare match, clear OC2A at Bottom (inverting mode)

Compare Output Mode, Fast PWM Mode

COM2B[1:0]	Description
00	Normal, OC2B disconnected
01	Reserved
10	Clear OC2B on compare match, set OC2B at Bottom (non-inverting mode)
11	Set OC2B on compare match, clear OC2B at Bottom (inverting mode)

Compare Output Mode, Phase Correct PWM

COM2A[1:0]	Description
00	Normal, OC2A disconnected

01	WGM22 = 0: normal operation, OC2A disconnected
	WGM22 = 1: Toggle OC2A on compare match

10	Clear OC2A on compare match, when upcounting. Set OC2A on compare match when down counting
11	Set OC2A on compare match, when upcounting. Set OC2A on compare match when down counting

Compare Output Mode, Phase Correct PWM

COM2B[1:0]	Description
00	Normal, OC2B disconnected
01	Reserved
10	Clear OC2B on compare match, when upcounting. Set OC2B on compare match when down counting
11	Set OC2B on compare match, when upcounting. Set OC2B on compare match when down counting

Figure 6.15: Timercounter control register A and B (TCCR2A and B) bit settings.

Timer/Counter Interrupt Flag Register 2 (TIFR2)

Timer 2 uses the Output Compare Flags 2 A and B (OCF2A and B) to include which an output compare match. Timer 2 also uses the Timer/Counter 2 Overflow Flag (TOV2) to detect Timer/Counter 2 overflows.

6.9 PROGRAMMING THE TIMER SYSTEM

In this section we present several representative examples of using the timer system for various applications. We will provide examples of using the timer system to generate a prescribed delay, to generate a PWM signal, and to capture an input event.

6.9.1 PRECISION DELAY

Example: In this example we program the ATmega164 to provide a delay of some number of 6.55 ms interrupts. The Timer 0 overflow is configured to occur every 6.55 ms. The overflow flag is used as a "clock tick" to generate a precision delay. To create the delay the microcontroller is placed in a while loop waiting for the prescribed number of Timer 0 overflows to occur. The delay is used to toggle an LED connected to PORTA[0].

```
//*****************************************************************
//flash_led.c
//*****************************************************************

#include <iom164pv.h>

//function prototypes*********************************************
void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);
void timer0_interrupt_isr(void);
void delay(unsigned int number_of_6_55ms_interrupts);
void initialize_ports(void);

//initialize timer0 overflow interrupt******************************
                                       //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:19


//global variables************************************************
unsigned int   input_delay;            //counts num of Timer/Counter0
                                       //Overflow interrupts
```

```
//main program*********************************************************

void main(void)
{
init_timer0_ovf_interrupt();  //initialize Timer/Counter0 Overflow
                              //interrupt - call once at beginning
initialize_ports();
PORTA = PORTA | 0x01;         //PORTD[0] high

while(1)
  {
  delay(153);                 //1 second delay
  PORTA = PORTA & 0xFE;       //PORTA[0] low

  delay(153);                 //1 second delay
  PORTA = PORTA | 0x01;       //PORTD[1] high
  }
}

//*********************************************************************
//int_timer0_ovf_interrupt(): The Timer/Counter0 Overflow interrupt is
//being employed as a time base for a master timer for this project. The
//ceramic resonator  operating at 10 MHz is divided by 256.  The 8-bit
//Timer0 register (TCNT0) overflows every 256 counts or every 6.55 ms.
//*********************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04;                //divide timer0 timebase by 256,
                              //overflow occurs every 6.55ms
TIMSK0 = 0x01;                //enable timer0 overflow interrupt
asm("SEI");                   //enable global interrupt
}

//********************************************************************
//********************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing the
//corresponding interrupt handling vector.
```

```
//**********************************************************************

void timer0_interrupt_isr(void)
{
input_delay++;               //increment overflow counter
}

//**********************************************************************
//delay(unsigned int num_of_6_55ms_interrupts): this generic delay
//function provides the specified delay as the number of 6.55 ms
//"clock ticks" from the Timer/Counter0 Overflow interrupt.
//Note: this function is only valid when using a 10 MHz crystal or
//ceramic resonator.
//**********************************************************************

void delay(unsigned int number_of_6_55ms_interrupts)
{
TCNT0 = 0x00;                   //reset timer0
input_delay = 0;                //reset timer0 overflow counter
while(input_delay <= number_of_6_55ms_interrupts)
  {
  ;                             //wait for number of interrupts
  }
}

//**********************************************************************
//initialize_ports: provides initial configuration for I/O ports
//**********************************************************************

void initialize_ports(void)
{
DDRA =0xff;  //set PORTA as output
PORTA=0x01;  //initialize low

DDRB =0x0a;  //PORTB[7:4] as input, set PORTB[3:0] as output
PORTB=0x00;  //disable PORTB pull-up resistors

DDRC =0xff; //set PORTC as output
PORTC=0x00; //initialize low
DDRD =0xff; //set PORTD as output
PORTD=0x00; //initialize low
}

//*********************************************************************
```

The delay function generates an interrupt every 6.55 ms when the microcontroller is clocked from a 10 MHz source. The interrupt period will change when the microcontroller is clocked at a different frequency.

6.9.2 PULSE WIDTH MODULATION

Example: In this example, PWM signals are generated on OC1A (PORTD[5]) and OC1B (PORTD[4]) pins. The ATmega164P is set for the internal 8 MHz oscillator with divide by eight fuze set resulting in an overall operating frequency of 1 MHz.

```
//*********************************************************************
//pwm1.c
// - Generates PWM signals on OC1A (PORTD[5]) and OC1B (PORTD[4])
// - Internal 8 MHz oscillator with divide-by-eight fuse set
// - Overall operating frequency 1 MHz
//*********************************************************************

#include <iom164pv.h>

//function prototypes***************************************************
void PWM_forward(void);
void initialize_ports(void);

void main(void)
{
initialize_ports();
PWM_forward();

while(1)
  {
  ;
  }

}
```

```
//**********************************************************************
//void PWM_forward(void): the PWM is configured to make the motors go
//forward.
//Implementation notes:
// - The left motor is controlled by PWM channel OC1B
// - The right motor is controlled by PWM channel OC1A
// - To go forward the same PWM duty cycle is applied to both the left
// and right motors.
//***********************************************************************

void PWM_forward(void)
{
TCCR1A = 0xA1;                    //freq = resonator/510 = 10 MHz/510
                                  //freq = 19.607 kHz
TCCR1B = 0x03;                    //no clock source division
                                  //Initiate PWM duty cycle variables
                                  //Set PWM for left and right motors
                                  //to 50
OCR1BH = 0x00;                    //PWM duty cycle CH B left motor
OCR1BL = (unsigned char)(128);
OCR1AH = 0x00;                    //PWM duty cycle CH B right motor
OCR1A = (unsigned char)(128);
}

//*********************************************************************
//initialize_ports: provides initial configuration for I/O ports
//*********************************************************************

void initialize_ports(void)
{
DDRA =0xff;  //set PORTA as output
PORTA=0x01;  //initialize low

DDRB =0x0a;  //PORTB[7:4] as input, set PORTB[3:0] as output
PORTB=0x00;  //disable PORTB pull-up resistors

DDRC =0xff; //set PORTC as output
PORTC=0x00; //initialize low
DDRD =0xff; //set PORTD as output
PORTD=0x00; //initialize low
}

//*********************************************************************
```

Example: In this example Timer 1, Channel B is used to generate a pulse width modulated signal on PORTD[4] (pin 18). An analog voltage provided to ADC Channel 3 PORTA[3] (pin 37) is used to set the desired duty cycle from 50–100% as shown in Figure 6.16.

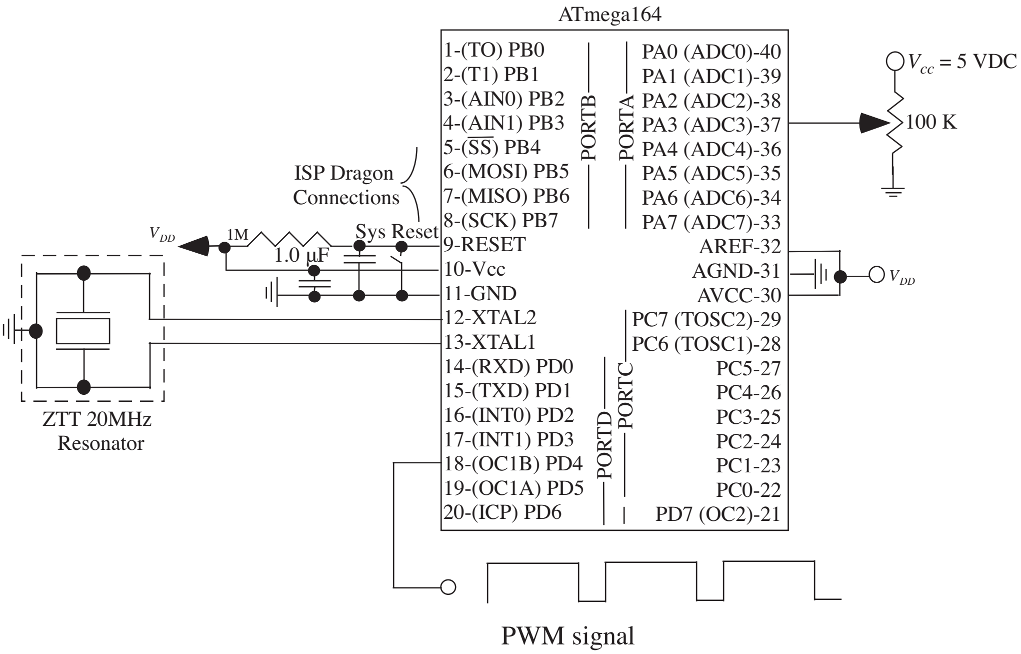


Figure 6.16: Pulse width modulation.

```
//**************************************************************
//pwm_adc_spd_cntrl.c: Timer 1, Channel B is used to generate
//PWM signal on PORTD[4] (pin 18).  An analog voltage provided
//to ADC Channel 3 PORTA[3] (pin 37) is used to set the desired
//duty cycle from 50 to 100
//**************************************************************

#include <iom164pv.h>
```

```
//function prototypes******************************************
void InitADC( void);
unsigned int ReadADC(unsigned char channel);
void initialize_ports(void);
void set_pwm_parameters(void);

//global variables*********************************************
unsigned int  speed_int;
float         speed_float;

//main program*************************************************

void main(void)
{
initialize_ports();
InitADC();                        //Initialize ADC

while(1)
  {
  set_pwm_parameters();
  }

}

//*************************************************************
//InitADC: initialize analog-to-digital converter
//*************************************************************
void InitADC( void)
{
ADMUX = 0;                      //Select channel 0
ADCSRA = 0xC3;                  //Enable ADC & start 1st
                                //dummy conversion
                                //Set ADC module prescalar
                                //to 8 critical for
                                //accurate ADC results
while (!(ADCSRA & 0x10));       //Check if conversion ready
ADCSRA |= 0x10;                 //Clear conv rdy flag -
                                //set the bit
}
```

```
//*************************************************************
//ReadADC: read analog voltage from analog-to-digital converter -
//the desired channel for conversion is passed in as an unsigned
//character variable. The result is returned as a left justified,
//10 bit binary result. The ADC prescalar must be set to 8 to
//slow down the ADC clock at higher external clock frequencies
//(10 MHz) to obtain accurate results.
//*************************************************************
unsigned int ReadADC(unsigned char channel)
{
unsigned int binary_weighted_voltage, binary_weighted_voltage_low;
unsigned int binary_weighted_voltage_high; //weighted binary
                                 //voltage
ADMUX = channel;                 //Select channel
ADCSRA |= 0x43;                  //Start conversion
                                 //Set ADC module prescalar
                                 //to 8 critical for
                                 //accurate ADC results
while (!(ADCSRA & 0x10));        //Check if conversion ready
ADCSRA |= 0x10;                  //Clear Conv rdy flag - set
                                 //the bit
binary_weighted_voltage_low = ADCL; //Read 8 low bits first
                                 //(important)
                                 //Read 2 high bits,
                                 //multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage = binary_weighted_voltage_low |
                          binary_weighted_voltage_high;
return binary_weighted_voltage;  //ADCH:ADCL
}

//*************************************************************
//*************************************************************
//initialize_ports: provides initial configuration for I/O ports
//*************************************************************

void initialize_ports(void)
{
DDRA =0xff;  //set PORTA as output
PORTA=0x01;  //initialize low

DDRB =0x0a;  //PORTB[7:4] as input, set PORTB[3:0] as output
PORTB=0x00;  //disable PORTB pull-up resistors

DDRC =0xff; //set PORTC as output
PORTC=0x00; //initialize low

DDRD =0xff; //set PORTD as output
PORTD=0x00; //initialize low
}

//**************************************************************
//void set_pwm_parameters(void): reads analog voltage on PORTA[3].
//Converts voltage to duty cycle:  0 VDC = 50
//**************************************************************


void set_pwm_parameters(void)
{
                                    //Read PWM duty cycle setting PA3
speed_int = ReadADC(0x03);          //speed setting unsigned int
                                    //Convert to max duty cycle setting
                                    //0 VDC = 50
speed_float = ((float)(speed_int)/(float)(0x0400));
                                    //Convert V to PWM constant 127-255
speed_int = (unsigned int)((speed_float * 127) + 128.0);
                                    //Configure PWM clock
TCCR1A = 0xA1;                      //freq = int osc/64 = 1 MHz/64
                                    //freq = 15.6 kHz
TCCR1B = 0x03;                      //no clock source division
                                    //Initiate PWM duty cycle variables
OCR1BH = 0x00;
OCR1BL = (unsigned char)(speed_int);//Set PWM duty cycle CH B to 0

}

//****************************************************************
```

In certain applications, it is useful to slowly ramp up to the desired PWM duty cycle. The following example illustrates how to do this.

Example: In this example a PWM duty cycle is set using the voltage provided to PORTA[3] (pin 37). The duty cycle starts at 0 and increments up to the set value in 1 s increments.

```
//*********************************************************************
//pwm.c: PWM duty cycle is set using the voltage provided to PORTA[3]
//(pin 37).  The duty cycle starts at 0 and increments up to the set
//value in 1 second increments.
//*********************************************************************

#include <iom164pv.h>

//function prototypes*************************************************
void InitADC( void);
unsigned int ReadADC(unsigned char channel);
void initialize_ports(void);
void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);
void timer0_interrupt_isr(void);

//initialize timer0 overflow interrupt********************************
                                  //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:19


//global variables****************************************************
unsigned int  open_speed_int;
float         open_speed_float;
int           gate_position_int;
unsigned int  PWM_open_incr = 1;
unsigned int  input_delay;        //counts num of Timer/Counter0
                                  //Overflow interrupts

//main program*********************************************************

int main(void)
{
unsigned char PWM_duty_cycle = 0;
init_timer0_ovf_interrupt();     //initialize Timer/Counter0 Overflow
                                 //interrupt - call once at beginning
initialize_ports();
InitADC();                       //Initialize ADC
                                 //Read "Open Speed" voltage at PA3
open_speed_int = ReadADC(0x03);  //Open Speed Setting unsigned int
                                 //Convert to max duty cycle setting
                                 //0 VDC = 50
open_speed_float = ((float)(open_speed_int)/(float)(0x0400));
                                 //Convert volt to PWM constant 127-255
open_speed_int = (unsigned int)((open_speed_float * 127) + 128.0);
                                 //Configure PWM clock
TCCR1A = 0xA1;                   //freq = int osc/64 = 1 MHz/64
                                 //freq = 15.6 kHz
TCCR1B = 0x03;                   //no clock source division
                                 //Initiate PWM duty cycle variables
PWM_duty_cycle = 0;

OCR1BH = 0x00;
OCR1BL = (unsigned char)(PWM_duty_cycle);//Set PWM duty cycle CH B to 0
                                         //Ramp up to Open Speed in 1.6s

while (PWM_duty_cycle < open_speed_int)
  {
  delay(153);                              //1 second delay
  if(PWM_duty_cycle < open_speed_int)      //Increment duty cycle
    PWM_duty_cycle=PWM_duty_cycle + PWM_open_incr;
  OCR1BL = (unsigned char)(PWM_duty_cycle);//Set PWM duty cycle CH B
  }

while(1)
  {
  ; //continue to generate PWM signal
  }

return 0;
}

//*************************************************************
```

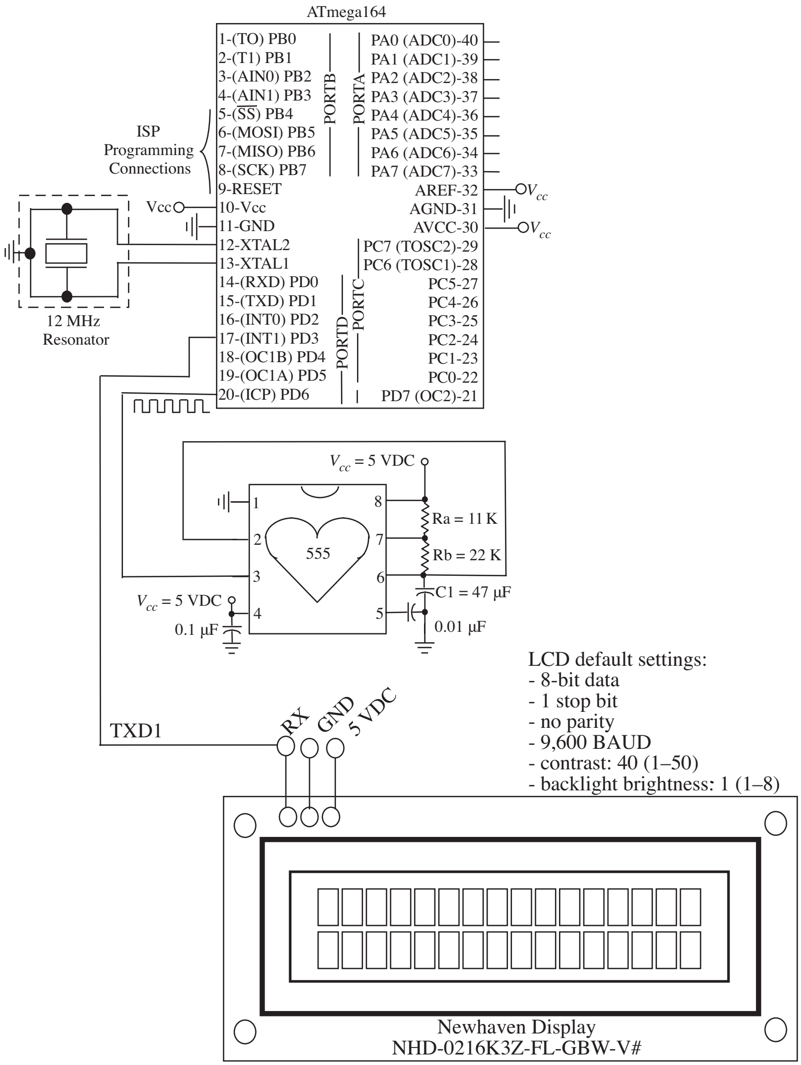
```
//InitADC: initialize analog-to-digital converter
//*************************************************************
void InitADC( void)
{
ADMUX = 0;                       //Select channel 0
ADCSRA = 0xC3;                   //Enable ADC & start 1st
                                 //dummy conversion
                                 //Set ADC module prescalar
                                 //to 8 critical for
                                 //accurate ADC results
while (!(ADCSRA & 0x10));        //Check if conversation is
                                 //ready
ADCSRA |= 0x10;                  //Clear conv rdy flag -
                                 //set the bit
}
//*************************************************************
//ReadADC: read analog voltage from analog-to-digital converter -
//the desired channel for conversion is passed in as an unsigned
//character variable. The result is returned as a left justified,
//10 bit binary result. The ADC prescalar must be set to 8 to
//slow down the ADC clock at higher external clock frequencies
//(10 MHz) to obtain accurate results.
//*************************************************************
unsigned int ReadADC(unsigned char channel)
{
unsigned int binary_weighted_voltage, binary_weighted_voltage_low;
unsigned int binary_weighted_voltage_high; //weighted binary
                                           //voltage
ADMUX = channel;                           //Select channel
ADCSRA |= 0x43;                            //Start conversion
                                           //Set ADC prescalar
                                           //to 8 critical for
                                           //accurate ADC results
while (!(ADCSRA & 0x10));                  //Check if conversion is
                                           //ready
ADCSRA |= 0x10;                            //Clear Conv rdy flag - set
                                           //the bit
binary_weighted_voltage_low = ADCL;        //Read 8 low bits first
                                           //(important)
                                           //Read 2 high bits,
                                           //multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage = binary_weighted_voltage_low |
                          binary_weighted_voltage_high;
return binary_weighted_voltage;            //ADCH:ADCL
}

//*************************************************************
//*************************************************************
//initialize_ports: provides initial configuration for I/O ports
//*************************************************************

void initialize_ports(void)
{
DDRA =0xff;  //set PORTA as output
PORTA=0x01;  //initialize low

DDRB =0x0a;  //PORTB[7:4] as input, set PORTB[3:0] as output
PORTB=0x00;  //disable PORTB pull-up resistors

DDRC =0xff;  //set PORTC as output
PORTC=0x00;  //initialize low

DDRD =0xff;  //set PORTD as output
PORTD=0x00;  //initialize low
}

//*************************************************************
//*************************************************************
//int_timer0_ovf_interrupt(): The Timer/Counter0 Overflow
//interrupt is being employed as a time base for a master timer
//for this project. The ceramic resonator  operating at 10 MHz
//is divided by 256.  The 8-bit Timer0 register (TCNT0) overflows
//every 256 counts or every 6.55 ms.
//*************************************************************

void init_timer0_ovf_interrupt(void)
{
```

```
TCCR0B = 0x04;                //divide timer0 timebase by 256,
                              //overflow occurs every 6.55ms
TIMSK0 = 0x01;                //enable timer0 overflow interrupt
asm("SEI");                   //enable global interrupt
}

//*************************************************************
//*************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing
//the corresponding interrupt handling vector.
//*************************************************************

void timer0_interrupt_isr(void)
{
input_delay++;                 //increment overflow counter
}

//*************************************************************
//delay(unsigned int num_of_6_55ms_interrupts): this generic
//delay function provides the specified delay as the number of
//6.55 ms "clock ticks" from the Timer/Counter0 Overflow
//interrupt.
//Note: this function is only valid when using a 10 MHz crystal
//or ceramic resonator.
//*************************************************************

void delay(unsigned int number_of_6_55ms_interrupts)
{
TCNT0 = 0x00;                   //reset timer0
input_delay = 0;                //reset timer0 overflow counter
while(input_delay <= number_of_6_55ms_interrupts)
  {
  ;                              //wait for number of interrupts
  }
}

//*********************************************************************
```

6.9.3 INPUT CAPTURE MODE

This input capture example is based on a design developed by Julie Sandberg, BSEE and Kari Fuller, BSEE at the University of Wyoming as part of their senior design project. In this example the input capture channel (PD6, ICP) is being used to monitor the heart rate (typically 50–120 beats per minute) of a patient. The microcontroller is set to operate at an internal clock frequency of 2.5 MHz (20 MHz ceramic resonator and fuse set for divide by eight). The 555 timer is used to simulate a heartbeat as shown in Figure 6.17. Results are displayed on a serial LCD module (Newhaven NHD-0216K3Z-FL-GBW-V3, www.newhavendisplay.com).

```
//*************************************************************
//file name: heartbeat3.c
//*************************************************************

//include files***********************************************

//MICROCHIP register definitions for ATmega164
#include <iom164pv.h>

//function prototypes*****************************************
void delay(unsigned int number_of_26_2ms_interrupts);
void init_timer0_ovf_interrupt(void);
void initialize_ports(void);             //initializes ports
void timer0_interrupt_isr(void);
void clear_LCD(void);
void calculate_trip_int(void);
void input_capture_ISR(void);
void initialize_ICP_interrupt(void);
void USART_init(void);
void USART_transmit(unsigned char data);
void LCD_init_serial(void);
void lcd_print_string(char str[]);
void move_LCD_cursor(unsigned char position);
void clear_LCD(void);
void print_heart_rate(void);
void heart_rate(void);

//interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:19
#pragma interrupt_handler input_capture_ISR:13
```



LCD default settings:
- 8-bit data
- 1 stop bit
- no parity
- 9,600 BAUD
- contrast: 40 (1–50)
- backlight brightness: 1 (1–8)

Prefix	Command	Description
--	--	Display character (0x00-0xff)
0xFE	0x41	Display on
0xFE	0x45	Position - line 1: 0x00–0x0F - line 2: 0x40–0x4F
0xFE	0x46	Cursor home
0xFE	0x50	Clear screen
0xFE	0x52	Set contrast (1–50)
0xFE	0x53	Set background brightness (1–8)

Figure 6.17: Heartbeat simulator.

```
//main program*************************************************

//global variables
//initialize all variables as zero
unsigned int delay_timer = 0;
unsigned int first_edge=0;
unsigned int time_pulses=0;
unsigned int time_pulses_low=0;
unsigned int time_pulses_high=0;
unsigned int HR=0;
unsigned int i;

void main(void)
{
initialize_ports();                   //initialize ports
USART_init();
LCD_init_serial();
init_timer0_ovf_interrupt();          //init Timer0 for delay

clear_LCD();

initialize_ICP_interrupt();           //init input capture int

delay(38);                            //delay 1s
PORTD = PORTD | 0x80;                 //LED ON PORTD[7]


while(1)
  {
  delay(190);                          //delay 5s
  print_heart_rate();
  }
}

//*************************************************************
//function definitions*****************************************

//*************************************************************
//initialize_ports: provides initial configuration for I/O ports
```

```
//***************************************************************

void initialize_ports(void)
{
DDRA =0xff;  //set PORTA as output
PORTA=0x00;  //initialize low

DDRB =0x0f;  //PORTB[7:4] as input, set PORTB[3:0] as output
PORTB=0x00;  //disable PORTB pull-up resistors

DDRC =0xff; //set PORTC as output
PORTC=0x00; //initialize low

DDRD =0xbf; //set PORTD as output, PORTD[6] input  1011_1111
PORTD=0x00; //initialize low
}

//***************************************************************
//delay(unsigned int num_of_26_2_ms_interrupts): this generic
//delay function provides the specified delay as the number of
//26.2 ms "clock ticks" from the Timer0 interrupt.
//
//Note: this function is only valid when using a 2.5 MHz
//time base.  e.g., 20 MHz ceramic resonator divided by 8
//***************************************************************

void delay(unsigned int number_of_26_2ms_interrupts)
{
TCNT0 = 0x00;                         //reset delay_timer
delay_timer = 0;
while(delay_timer <= number_of_26_2ms_interrupts)
  {
  ;
  }
}

//***************************************************************
//int_timer0_ovf_interrupt(): The Timer0 overflow interrupt is
//being employed as a time base for a master timer for this
```

```
//project. The internal time base is set to operate at 2.5 MHz and
//then is divided by 256.  The 8-bit Timer0 register (TCNT0)
//overflows every 256 counts or every 26.2 ms.
//***************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04; //divide timer0 timebase by 256, overflow occurs
               //every 26.2ms
TIMSK0 = 0x01; //enable timer0 overflow interrupt
asm("SEI");    //enable global interrupt
}


//***************************************************************
//void timer0_interrupt_isr(void)
//***************************************************************

void timer0_interrupt_isr(void)
{
delay_timer++;                            //increment timer
}

//***************************************************************
//initialize_ICP_interrupt: Initialize Timer/Counter 1 for
//input capture
//***************************************************************

void initialize_ICP_interrupt(void)
{
TIMSK1=0x20;       //Allows input capture interrupts
TCCR1A=0x00;       //No output comp or waveform generation mode
TCCR1B=0x45;       //Capture on rising edge, clock prescalar=1024
TCNT1H=0x00;       //Initially clear timer/counter 1
TCNT1L=0x00;
asm("SEI");        //Enable global interrupts
}

//****************************************************************
```

```
void input_capture_ISR(void)
{
//toggle LED PORTD[7]
PORTD ^= 0x80;     //toggle LED PORTD[7]

if(first_edge==0)
  {
  ICR1L=0x00;      //Clear ICR1 and TCNT1 on first edge
  ICR1H=0x00;
  TCNT1L=0x00;
  TCNT1H=0x00;
  first_edge=1;
  }

else
  {
  ICR1L=TCNT1L;    //Capture time from TCNT1
  ICR1H=TCNT1H;
  TCNT1L=0x00;
  TCNT1H=0x00;
  first_edge=0;
  }

heart_rate();      //Calculate the heart rate
TIFR1=0x20;        //Clear the input capture flag
asm("RETI");       //Resets I flag to allow global interrupts
}

//****************************************************************
//void heart_rate(void)
//Note:  Fosc = 2.5 MHz (20 MHz ceramic resonator, divide by 8
//       TCCR1B set for divide by 1024
//****************************************************************

void heart_rate(void)
{
if(first_edge==0)
  {
```

```
  time_pulses_low = ICR1L;       //Read 8 low bits first
  time_pulses_high = ((unsigned int)(ICR1H << 8));
  time_pulses = time_pulses_low | time_pulses_high;
  if(time_pulses!=0)             //1 counter increment = 0.41 ms
    {                            //Divide by 2439 to get seconds/pulse
    HR=60/(time_pulses/2439);    //(secs/min)/(secs/beat) =bpm
    }
  else
    {
    HR=0;
    }
  }
  else
  {
  HR=0;
  }
}

//*************************************************************
//USART_init: initializes the USART system
//*************************************************************

void USART_init(void)
{
UCSR1A = 0x00; //control register initialization
UCSR1B = 0x08; //enable transmitter
UCSR1C = 0x86; //async, no parity, 1 stop bit,
//8 data bits
//Baud Rate initialization
//fosc = 2.5 MHz, 9600 BAUD rate
UBRR1H = 0x00; UBRR1L = 0x0f;
}

//*************************************************************
//USART_transmit: transmits single byte of data
//*************************************************************

void USART_transmit(unsigned char data)
{
```

```
while((UCSR1A & 0x20)==0x00) //wait for UDRE flag
{
;
}
UDR1 = data; //load data to UDR1 for transmission
}

//*************************************************************
//LCD_init: initializes the USART system
//*************************************************************

void LCD_init_serial(void)
{
USART_transmit(0xFE);
USART_transmit(0x41); //LCD on
USART_transmit(0xFE);
USART_transmit(0x46); //cursor to home
}

//*************************************************************
//void lcd_print_string(char str[])
//*************************************************************

void lcd_print_string(char str[])
{
int k = 0;

while(str[k] != 0x00)
  {
  USART_transmit(str[k]);
  k = k+1;
  delay(1);
  }
}

//*************************************************************
//void move_LCD_cursor(unsigned char position)
//*************************************************************
```

```
void move_LCD_cursor(unsigned char position)
{
USART_transmit(0xFE);
USART_transmit(0x45);
USART_transmit(position);
}

//*************************************************************
//void clear_LCD(void)
//*************************************************************

void clear_LCD(void)
{
USART_transmit(0xFE);
USART_transmit(0x51);
}

//*************************************************************
//void print_heart_rate(void)
//*************************************************************

void print_heart_rate(void)
{
int hundreths_place, tens_place, ones_place;
char hundreths_place_char, tens_place_char, ones_place_char;

clear_LCD();

//move cursor to line 1, position 0
move_LCD_cursor(0x00);
lcd_print_string("Heart rate:");

//print HR
//move cursor to line 2, position 0
move_LCD_cursor(0x40);

hundreths_place = HR/100;              //isolate first digit
                                       //convert to ascii
hundreths_place_char = (char)(hundreths_place + 48);
```

```
USART_transmit(hundreths_place_char);  //display first digit

                                       //isolate tens place
tens_place = (int)((HR - (hundreths_place*100))/10);
tens_place_char=(char)(tens_place+48); //convert to ASCII
USART_transmit(tens_place_char);       //print to LCD

                                       //isolate ones place
ones_place = (int)((HR - (hundreths_place*100))
ones_place_char=(char)(ones_place+48); //convert to ASCII
USART_transmit(ones_place_char);       //print to LCD
}

//***************************************************************
```

6.10 SERVO MOTOR CONTROL WITH THE PWM SYSTEM

A servo motor provides an angular displacement from 0–180°. Most servo motors provide the angular displacement relative to the pulse length of repetitive pulses sent to the motor, as shown in Figure 6.18. A 1 ms pulse provides an angular displacement of 0° while a 2 ms pulse provides a displacement of 180°. Pulse lengths in between these two extremes provide angular displacements between 0 and 180°. Usually a 20–30 ms low signal is provided between the active pulses.

A test and interface circuit for a servo motor is provided in Figure 6.18. The PB0 and PB1 inputs of the ATmega164 provide for clockwise (CW) and counter-clockwise (CCW) rotation of the servo motor, respectively. The time base for the ATmega164 is set for the 128 KHz internal RC oscillator. Also, the internal time base divide-by-eight circuit is active via a fuse setting. Pulse width modulated signals to rotate the servo motor is provided by the ATmega164. A voltage-follower op amp circuit is used as a buffer between the ATmega164 and the servo motor. Use of an external ceramic resonator at 128 KHz is recommended for this application.

The software to support the test and interface circuit is provided below.

```
//********************************************************************
//target controller: MICROCHIP ATmega164
//
//Microchip AVR ATmega164P Controller Pin Assignments
//Chip Port Function I/O Source/Dest Asserted Notes
//PORTB:
//Pin 1 PB0 to active high RC debounced switch - CW
//Pin 2 PB1 to active high RC debounced switch - CCW
//Pin 9 Reset - 1M resistor to Vcc, tact switch to ground,
//      1.0 uF to ground
```

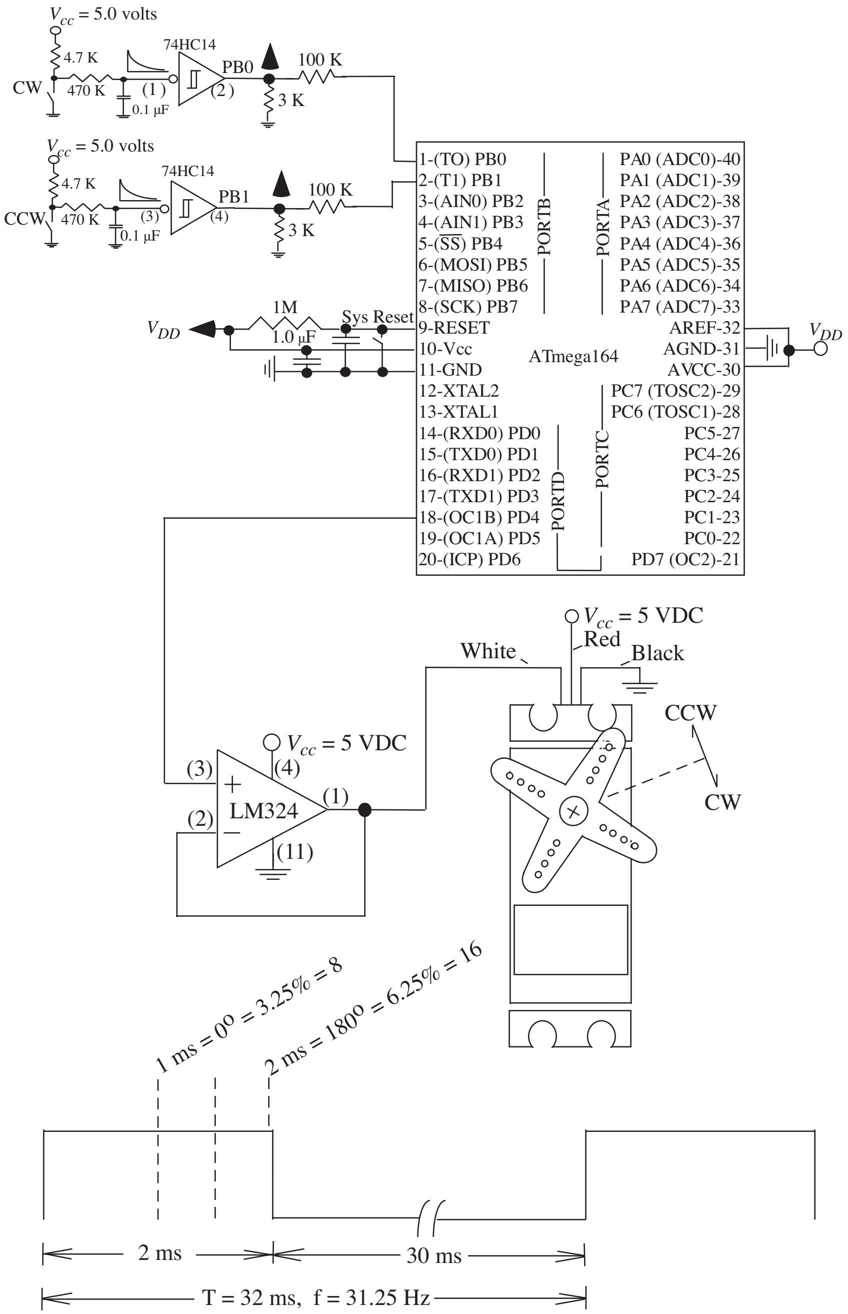


Figure 6.18: Test and interface circuit for a servo motor.

```
//Pin 10 Vcc - 1.0 uF to ground //Pin 11 Gnd
//Pin 12 ZTT-10.00MT ceramic resonator connection
//Pin 13 ZTT-10.00MT ceramic resonator connection
//Pin 18 PD4 - to servo control input //Pin 30 AVcc to Vcc
//Pin 31 AGnd to Ground //Pin 32 ARef to Vcc
//******************************************************************

//include files*****************************************************
//MICROCHIP register definitions for ATmega164

#include<iom164pv.h>
#include<macros.h>

//function
prototypes***********************************************************
void initialize_ports(void);           //initializes ports
void power_on_reset(void);             //return to startup state
void read_new_input(void);             //read input change on PORTB
void init_timer0_ovf_interrupt(void);  //initialize timer0 overflow
void InitUSART(void); void USART_TX(unsigned char data);

//main program*******************************************************
//The main program checks PORTB for user input activity.
//If new activity is found, the program responds.

//global variables
unsigned char   old_PORTB = 0x08;      //present value of PORTB
unsigned char   new_PORTB;             //new values of PORTB
unsigned int    PWM_duty_cycle;

void main(void)
{
power_on_reset();
initialize_ports();
InitUSART();
                                       //internal clock set for 128 KHZ
                                       //fuse set for divide by 8
                                       //configure PWM clock
TCCR1A = 0xA1;                         //freq = oscillator/510 =
                                       //128KHz/8/510
                                       //freq = 31.4 Hz
TCCR1B = 0x01;                         //no clock source division
                                       //duty cycle will vary from 3.1
                                       //1 ms = 0 degrees = 8 counts to
                                       //6.2

                                       //initiate PWM duty cycle
                                       //variables
PWM_duty_cycle = 12;
OCR1BH = 0x00;
OCR1BL = (unsigned char)(PWM_duty_cycle);

//main activity loop - processor will continually cycle through
//loop for new activity.  Activity initialized by external signals
//presented to PORTB[1:0]

while(1)
  {
  _StackCheck();                       //check for stack overflow
  read_new_input();                    //read input changes on PORTB
  }
}//end main

//Function definitions
//*******************************************************************
//power_on_reset:
//*******************************************************************

void power_on_reset(void)
{
initialize_ports();                    //initialize ports
}

//*******************************************************************
//initialize_ports: provides initial configuration for I/O ports
//*******************************************************************

void initialize_ports(void)
```

```
{
//PORTA
DDRA=0xff;                                      //PORTA[7:0] output
PORTA=0x00;                                     //Turn off pull ups

//PORTB DDRB=0xfc;
//PORTB[7-2] output, PORTB[1:0] input
PORTB=0x00;                                     //disable pull-up resistors

//PORTC DDRC=0xff;                              //set PORTC[7-0] as output
PORTC=0x00;                                     //init low

//PORTD
DDRD=0xff;                                      //set PORTD[7-0] as output
PORTD=0x00;                                     //initialize low }

//*********************************************************************
//*********************************************************************
//read_new_input: functions polls PORTB for a change in status.
//If status change has occurred, appropriate function for status
//change is called.
//Pin 1 PB0 to active high RC  debounced switch - CW
//Pin 2 PB1 to active high RC debounced switch - CCW
//********************************************************************

void read_new_input(void)
{
new_PORTB = (PINB);
if(new_PORTB != old_PORTB)
  {
  switch(new_PORTB)
    {                   //process change in PORTB input
    case 0x01:          //CW
      while(PINB == 0x01)
        {
        PWM_duty_cycle = PWM_duty_cycle + 1;
        if(PWM_duty_cycle > 16) PWM_duty_cycle = 16;
        OCR1BH = 0x00;
        OCR1BL = (unsigned char)(PWM_duty_cycle);
        }
      break;

    case 0x02:          //CCW
      while(PINB == 0x02)
        {
        PWM_duty_cycle = PWM_duty_cycle - 1;
        if(PWM_duty_cycle < 8) PWM_duty_cycle = 8;
        OCR1BH = 0x00;
        OCR1BL = (unsigned char)(PWM_duty_cycle);
        }
       break;

     default:;                         //all other cases
    }                                  //end switch(new_PORTB)
  }                                    //end if new_PORTB
  old_PORTB=new_PORTB;                 //update PORTB
}

//****************************************************************
```

6.11 SUMMARY

In this chapter, we considered a microcontroller timer system, associated terminology for timer related topics, discussed typical functions of a timer subsystem, studied timer hardware operations, and considered some applications where the timer subsystem of a microcontroller can be used. We then took a detailed look at the timer subsystem aboard the ATmega164 and reviewed the features, operation, registers, and programming of the three timer channels. We concluded with an example employing a servo motor.

6.12 REFERENCES AND FURTHER READING

Atmel 8-bit AVR Microcontroller with 16/32/64/128K Bytes In-System Programmable Flash, ATmega164A, ATmega164PA, ATmega324A, ATmega324PA, ATmega644A, ATmega644PA, ATmega1284, ATmega1284P, 8272C-AVR-06/11, data sheet: 8272C-AVR-06/11, Atmel, San Jose, CA, 2015.

Barrett, S. F. and Pack, D. J. *Microcontrollers Fundamentals for Engineers and Scientists*. Morgan & Claypool Publishers, 2006. DOI: 10.2200/s00025ed1v01y200605dcs001.

Driscoll, F., Coughlin, R., and Villanucci, R. *Data Acquisition and Process Control with the M68HC11 Microcontroller*, 2nd ed., Prentice Hall, Upper Saddle River, 2000.

Kenneth, S. *Embedded Microprocessor Systems Design: An Introduction Using the INTEL 80C188EB*. Prentice Hall, Upper Saddle River, NJ, 1998.

Morton, T. *Embedded Microcontrollers*. Prentice Hall, Upper Saddle River, NJ, 2001.

6.13 CHAPTER PROBLEMS

1. Given an 8-bit free-running counter and the system clock rate of 24 MHz, find the time required for the counter to count from zero to its maximum value.

2. If we desire to generate periodic signals with periods ranging from 125 ns–500 μs, what is the minimum frequency of the system clock if an 8-bit free-running counter is used?

3. The delay function generates an interrupt every 6.55 ms, when the microcontroller is clocked from a 10 MHz source. How would the function operate when the microcontroller is clocked from a 1 MHz source? 20 MHz source?

4. Describe how you can compute the period of an incoming signal with varying duty cycles.

5. Describe how one can generate an aperiodic pulse with a pulse width of 2 min?

6. Program the output compare system of the ATmega164 to generate a 1 kHz signal with a 10% duty cycle.

7. Design a microcontroller system to control a sprinkler controller that performs the following tasks. We assume that your microcontroller runs with 10 MHz clock and it has a 16-bit free-running counter. The sprinkler controller system controls two different zones by turning sprinklers within each zone on and off. To turn on the sprinklers of a zone, the controller needs to receive a 152.589 Hz PWM signal from your microcontroller. To turn off the sprinklers of the same zone, the controller needs to receive the PWM signal with a different duty cycle.

   (a) Your microcontroller needs to provide the PWM signal with 10% duty cycle for 10 ms to turn on the sprinklers in zone one.

   (b) After 15 min, your microcontroller must send the PWM signal with 15% duty cycle for 10 ms to turn off the sprinklers in zone one.

   (c) After 15 min, your microcontroller must send the PWM signal with 20% duty cycle for 10 ms to turn on the sprinklers in zone two.

   (d) After 15 min, your microcontroller must send the PWM signal with 25% duty cycle for 10 ms to turn off the sprinklers in zone two.

8. Modify the servo motor example to include a potentiometer connected to PORTA[0]. The servo will deflect 0° for 0 VDC applied to PORTA[0] and 180° for 5 VDC.

9. For the automated cooling fan example, what would be the effect of changing the PWM frequency applied to the fan?

10. Modify the code of the automated cooling fan example to also display the set threshold temperature.

CHAPTER 7

Microchip AVR® Operating Parameters and Interfacing

Objectives: After reading this chapter, the reader should be able to:

- describe the voltage and current parameters for the Microchip AVR HC CMOS-type microcontroller;
- apply the voltage and current parameters toward properly interfacing I/O devices to the Microchip AVR microcontroller;
- interface a wide variety of I/O devices to the Microchip AVR microcontroller;
- describe the special process that must be followed when the Microchip AVR microcontroller is used to interface to a high-power DC or AC device;
- explain the requirement for an optical-based interface;
- illustrate the procedure to control the speed and direction of a DC motor; and
- describe how to control several types of AC loads.

With Morgan & Claypool Publishers permission, we repeat in this chapter fundamental concepts of operating parameters and interfacing that first appeared in our first textbook for M&C, *Microcontrollers Fundamentals for Engineers and Scientists* [Barrett and Pack, 2006]. Information specific for the Microchip AVR line of microcontrollers and interface techniques for a number of additional I/O devices are also added.

We begin by reviewing the voltage and current electrical parameters for the HC CMOS-based Microchip AVR line of microcontrollers. We then show how to apply this information to properly interface I/O devices to the ATmega164 microcontroller. We then discuss the special considerations for controlling a high-power DC or AC load such as a motor and introduce the concept of an optical interface. Throughout the chapter, we provide a number of detailed examples.

7.1 OPERATING PARAMETERS

The microcontroller is an electronic device that has precisely defined operating conditions. As long as the microcontroller is used within its defined operating parameter limits, it should continue to operate correctly. However, if the allowable conditions are violated, spurious results or microcontroller damage may result.

Any time a device is connected to a microcontroller, careful interface analysis must be performed. Most microcontrollers are members of the "HC," or high-speed CMOS family of chips. As long as all components in a system are also of the "HC" family, as is the case for the Microchip AVR line of microcontrollers, electrical interface issues are minimal. If the microcontroller is connected to some component not in the HC family, electrical interface analysis must be completed. Manufacturers readily provide the electrical characteristic data necessary to complete this analysis in their support documentation.

To perform the interface analysis, there are eight different electrical specifications required for electrical interface analysis. The electrical parameters are:

- V_{OH}: the lowest guaranteed output voltage for a logic high,
- V_{OL}: the highest guaranteed output voltage for a logic low,
- I_{OH}: the output current for a V_{OH} logic high,
- I_{OL}: the output current for a V_{OL} logic low,
- V_{IH}: the lowest input voltage guaranteed to be recognized as a logic high,
- V_{IL}: the highest input voltage guaranteed to be recognized as a logic low,
- I_{IH}: the input current for a V_{IH} logic high, and
- I_{IL}: the input current for a V_{IL} logic low.

These electrical characteristics are required for both the microcontroller and the external components. Typical values for a microcontroller in the HC CMOS family assuming $V_{DD} = 5.0$ V and $V_{SS} = 0$ V are provided below. The minus sign on several of the currents indicates a current flow out of the device. A positive current indicates current flow into the device.

- $V_{OH} = 4.2$ V,
- $V_{OL} = 0.4$ V,
- $I_{OH} = -0.8$ mA,
- $I_{OL} = 1.6$ mA,
- $V_{IH} = 3.5$ V,

- $V_{IL} = 1.0$ V,
- $I_{IH} = 10\ \mu$A, and
- $I_{IL} = -10\ \mu$A.

It is important to realize that these are static values taken under very specific operating conditions. If external circuitry is connected such that the microcontroller acts as a current source (current leaving microcontroller) or current sink (current entering microcontroller), the voltage parameters listed above will also be affected.

In the current source case, an output voltage V_{OH} is provided at the output pin of the microcontroller when the load connected to this pin draws a current of I_{OH}. If a load draws more current from the output pin than the I_{OH} specification, the value of V_{OH} is reduced. If the load current becomes too high, the value of V_{OH} falls below the value of V_{IH} for the subsequent logic circuit stage and not be recognized as an acceptable logic high signal. When this situation occurs, erratic and unpredictable circuit behavior results.

In the sink case, an output voltage V_{OL} is provided at the output pin of the microcontroller when the load connected to this pin delivers a current of I_{OL} to this logic pin. If a load delivers more current to the output pin of the microcontroller than the I_{OL} specification, the value of V_{OL} increases. If the load current becomes too high, the value of V_{OL} rises above the value of V_{IL} for the subsequent logic circuit stage and not be recognized as an acceptable logic low signal. As before, when this situation occurs, erratic and unpredictable circuit behavior results.

For convenience, this information is illustrated in Figure 7.1. In Figure 7.1a, we have provided an illustration of the direction of current flow from the HC device and a comparison of voltage levels. As a reminder, current flowing out of a device is considered a negative current (source case); whereas, current flowing into the device is considered positive current (sink case). The magnitude of the voltage and current for HC CMOS devices are shown in Figure 7.1b. As more current is sunk or sourced from a microcontroller pin, the voltage will be pulled up or pulled down, respectively, as shown in Figure 7.1c. If I/O devices are improperly interfaced to the microcontroller, these loading conditions may become excessive, and voltages will not be properly interpreted as the correct logic levels.

You must also ensure that total current limits for an entire microcontroller port and overall bulk port specifications are complied with. For planning purposes, the sum of current sourced or sunk from a port should not exceed 100 mA. Furthermore, the sum of currents for all ports should not exceed 200 mA. As before, if these guidelines are not complied with, erratic microcontroller behavior or damage may result.

The procedures presented in the following sections when followed carefully will ensure the microcontroller will operate within its designed envelope. The remainder of the chapter is divided into input device interface analysis followed by output device interface analysis.

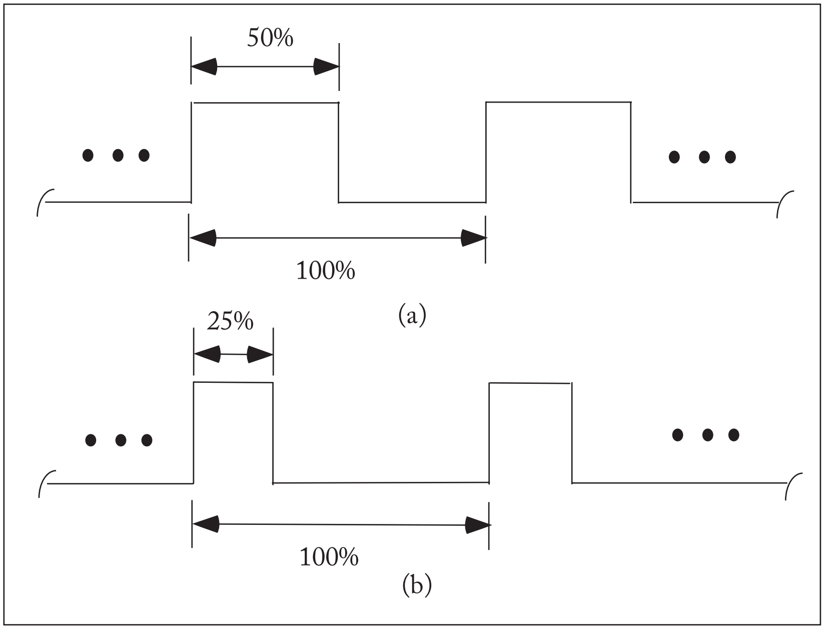


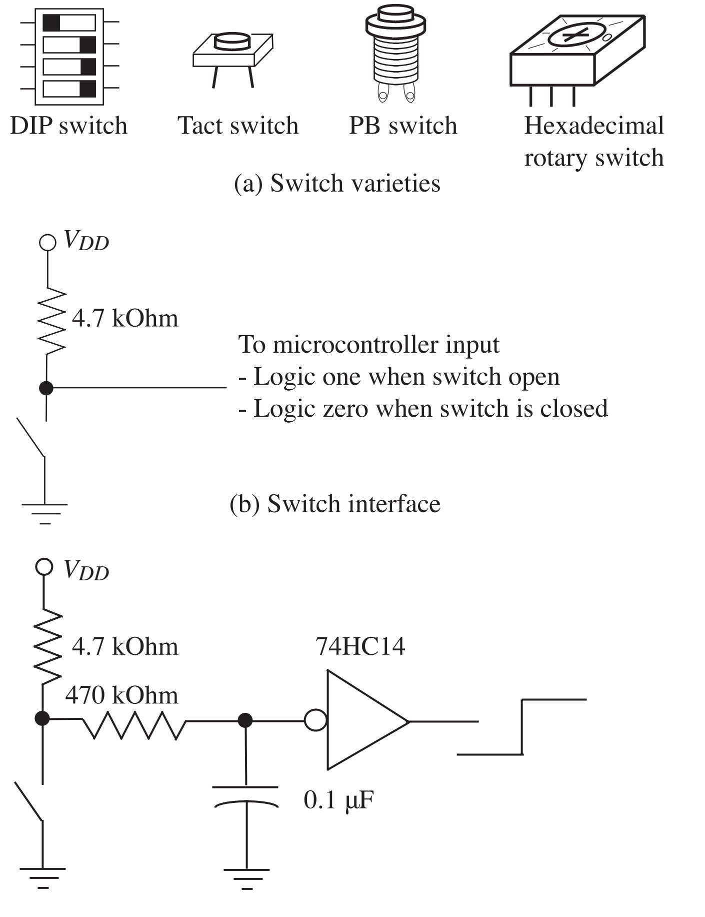
Figure 7.1: Electrical voltage and current parameters: (a) voltage and current electrical parameters, (b) HC CMOS voltage and current parameters, and (c) CMOS loading curves.

7.2 INPUT DEVICES

In this section, we discuss how to properly interface input devices to a microcontroller. We will start with the most basic input component, a simple on/off switch.

7.2.1 SWITCHES

Switches come in a variety of types. As a system designer, it is up to you to choose the appropriate switch for your specific application. Switch varieties commonly used in microcontroller applications are illustrated in Figure 7.2a. Here is a brief summary of the different types:



(a) Switch varieties

(b) Switch interface

(c) Switch interface equipped with debouncing circuitry

Figure 7.2: Switch interface: (a) switch varieties, (b) switch interface, and (c) switch interface equipped with debouncing circuitry.

- **Slide switch:** A slide switch has two different positions: on and off. The switch is manually moved to one position or the other. For microcontroller applications, slide switches are available that fit in the profile of a common integrated circuit size dual inline package (DIP). A bank of four or eight DIP switches in a single package is commonly available.

- **Momentary contact push-button switch:** A momentary contact push-button switch comes in two varieties: normally closed (NC) and normally open (NO). A NO switch, as its name implies, does not normally provide an electrical connection between its contacts. When the push-button portion of the switch is depressed the connection between the two switch contacts is made. The connection is held as long as the switch is depressed. When the switch is released, the connection is opened. The converse is true for an NC switch. For microcontroller applications, push-button switches are available in a small tact type switch configuration.

- **Push on/push off switches:** These type of switches are also available in an NO or NC configuration. For the NO configuration, the switch is depressed to make connection between the two switch contacts. The push button must be depressed again to release the connection.

- **Hexadecimal rotary switches:** Small profile rotary switches are available for microcontroller applications. These switches commonly have 16 rotary switch positions. As the switch is rotated to each position a unique 4-bit binary code is provided at the switch contacts.

A common switch interface is shown in Figure 7.2b. This interface allows a logic 1 or 0 to be properly introduced to a microcontroller input port pin. The basic interface consists of the switch in series with a current limiting resistor. The node between the switch and the resistor is provided to the microcontroller input pin. In the configuration shown, the resistor pulls the microcontroller input up to the supply voltage V_{DD}. When the switch is closed, the node is grounded, and a logic 0 is provided to the microcontroller input pin. To reverse the logic of the switch configuration, the position of the resistor and the switch is simply reversed.

7.2.2 SWITCH DEBOUNCING

Mechanical switches do not make a clean transition from one position (on) to another (off). When a switch is moved from one position to another, it makes and breaks contact multiple times. This activity may go on for tens of milliseconds. A microcontroller is relatively fast as compared with the action of the switch. Therefore, the microcontroller is able to recognize each switch bounce as a separate and erroneous transition.

To correct the switch bounce phenomena, additional external hardware components may be used or software techniques may be employed. A hardware debounce circuit is illustrated in Figure 7.2c. The node between the switch and the limiting resistor of the basic switch circuit is fed to a low pass filter (LPF) formed by the 470-kΩ resistor and the capacitor. The LPF prevents abrupt changes (bounces) in the input signal from the microcontroller. The LPF is followed by a 74HC14 Schmitt Trigger, which is simply an inverter equipped with hysteresis. Hysteresis provides for a different threshold when transitioning from logic high to logic low and from logic low to logic high. This provides some level of noise immunity about the threshold switching value and hence limits switch bouncing phenomena.

Switches may also be debounced using software techniques. This is accomplished by inserting a 30–50 ms lockout delay in the function responding to port pin changes. The delay prevents the microcontroller from responding to the multiple transitions related to switch bouncing.

You must carefully analyze a given design to determine if hardware or software switch debouncing techniques will be used. It is important to remember that all switches exhibit bounce phenomena and therefore must be debounced.

7.2.3 KEYPADS

A keypad is simply an extension of the simple switch configuration. A typical keypad configuration and interface are shown in Figure 7.3. As you can see, the keypad is simply multiple switches in the same package. A hexadecimal keypad is provided in the figure. A single row of keypad switches is asserted by the microcontroller, and then the host keypad port is immediately read. If a switch has been depressed, the keypad pin corresponding to the column the switch is in will also be asserted. The combination of a row and a column assertion can be decoded to determine which key has been pressed as illustrated in the table. Keypad rows are continually asserted one after the other in sequence. Because the keypad is a collection of switches, debounce techniques must also be employed.

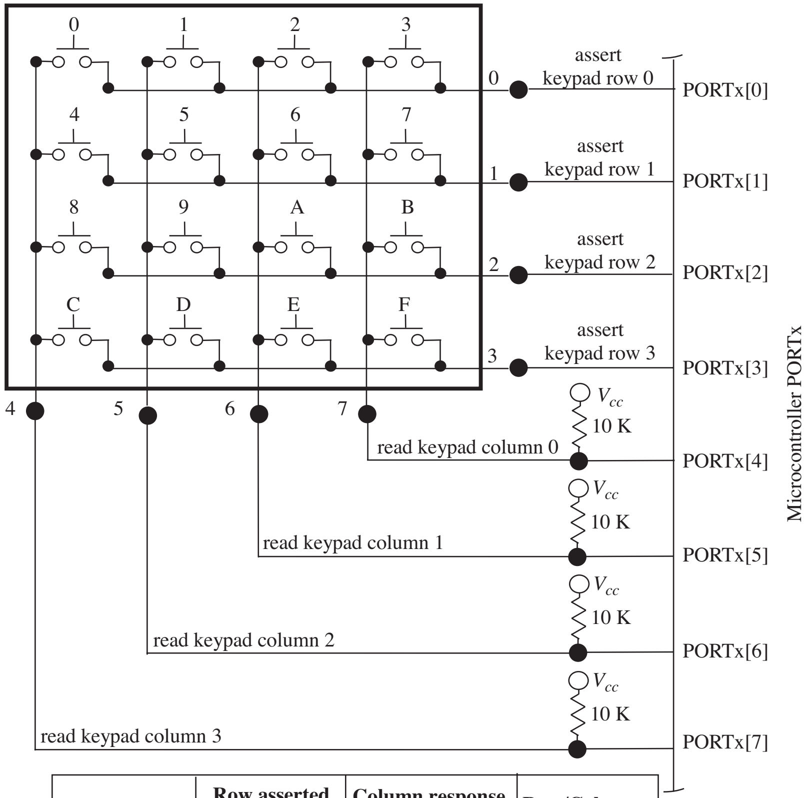
The keypad may be used to introduce user requests to a microcontroller. A standard keypad with alphanumeric characters may be used to provide alphanumeric values to the microcontroller such as providing your personal identification number (PIN) for a financial transaction. However, some keypads are equipped with removable switch covers such that any activity can be associated with a key press.

7.2.4 SENSORS

A microcontroller is typically used in control applications where data are collected, assimilated, and processed by the host algorithm and a control decision and accompanying signals are provided by the microcontroller. Input data for the microcontroller are collected by a complement of input sensors. These sensors may be digital or analog in nature.

Digital Sensors

Digital sensors provide a series of digital logic pulses with sensor data encoded. The sensor data may be encoded in any of the parameters associated with the digital pulse train such as duty cycle,

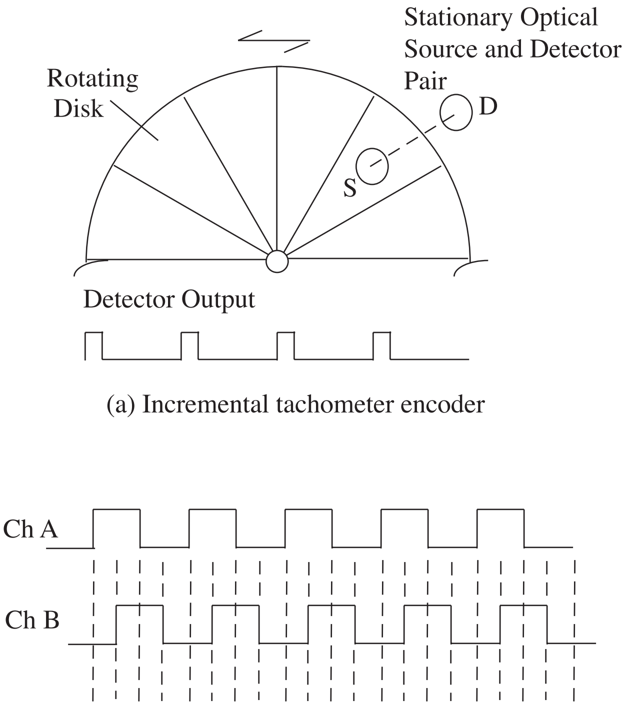


Key pressed by user	Row asserted by microcontroller				Column response from keypad switch				Row/Column combination read at micro port
	3	2	1	0	7	6	5	4	
0	1	1	1	0	1	1	1	0	0xEE
1	1	1	1	0	1	1	0	1	0xDE
2	1	1	1	0	1	0	1	1	0xBE
3	1	1	1	0	0	1	1	1	0x7E
4	1	1	0	1	1	1	1	0	0xED
5	1	1	0	1	1	1	0	1	0xDD
6	1	1	0	1	1	0	1	1	0xBD
7	1	1	0	1	0	1	1	1	0x7D
8	1	0	1	1	1	1	1	0	0xEB
9	1	0	1	1	1	1	0	1	0xDB
A	1	0	1	1	1	0	1	1	0xBB
B	1	0	1	1	0	1	1	1	0x7B
C	0	1	1	1	1	1	1	0	0xE7
D	0	1	1	1	1	1	0	1	0xD7
E	0	1	1	1	1	0	1	1	0xB7
F	0	1	1	1	0	1	1	1	0x77
none	X	X	X	X	1	1	1	1	0xXF

Figure 7.3: Keypad interface.

frequency, period, or pulse rate. The input portion of the timing system may be configured to measure these parameters.

An example of a digital sensor is the optical encoder. An optical encoder consists of a small plastic transparent disk with opaque lines etched into the disk surface. A stationary optical emitter and detector source are placed on either side of the disk. As the disk rotates, the opaque lines break the continuity between the optical source and detector. The signal from the optical detector is monitored to determine disk rotation as shown in Figure 7.4.



(a) Incremental tachometer encoder

(b) Incremental quadrature encoder

Figure 7.4: Optical encoder: (a) incremental tachometer encoder and (b) incremental quadrature encoder.

Optical encoders are available in a variety of types, depending on the information desired. There are two major types of optical encoders: incremental and absolute encoders. An absolute encoder is used when it is required to retain position information when power is lost. For example, if you were using an optical encoder in a security gate control system, an absolute encoder would be used to monitor the gate position. An incremental encoder is used in applications where a velocity or a velocity and direction information is required.

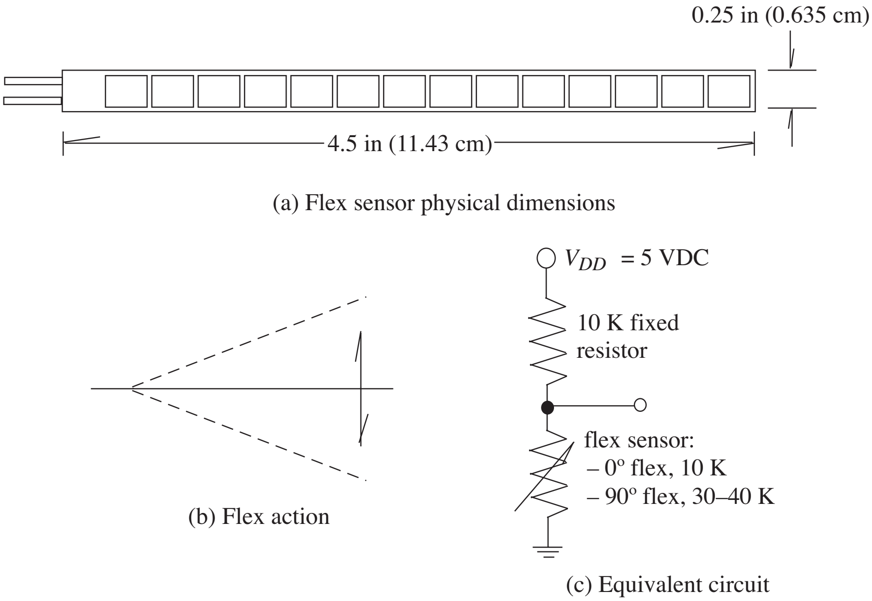
The incremental encoder types may be further subdivided into tachometers and quadrature encoders. An incremental tachometer encoder consists of a single track of etched opaque lines, as shown in Figure 7.4a. It is used when the velocity of a rotating device is required. To calculate velocity, the number of detector pulses is counted in a fixed amount of time. Because the number of pulses per encoder revolution is known, velocity may be calculated.

The quadrature encoder contains two tracks shifted in relationship to one another by 90°. This allows the calculation of both velocity and direction. To determine direction one would monitor the phase relationship between Channel A and Channel B, as shown in Figure 7.4b. The absolute encoder is equipped with multiple data tracks to determine the precise location of the encoder disk [Sick Stegmann].

Analog Sensors

Analog sensors provide a DC voltage that is proportional to the physical parameter being measured. As discussed in the ADC chapter, the analog signal may be first preprocessed by external analog hardware such that it falls within the voltage references of the conversion subsystem. The analog voltage is then converted to a corresponding binary representation.

An example of an analog sensor is the flex sensor shown in Figure 7.5a. The flex sensor provides a change in resistance for a change in sensor flexure. At 0° flex, the sensor provides 10



(a) Flex sensor physical dimensions

(b) Flex action

(c) Equivalent circuit

Figure 7.5: Flex sensor: (a) flex sensor's physical dimensions, (b) flex action, and (c) equivalent circuit.

kΩ of resistance. For 90° flex, the sensor provides 3040 kΩ of resistance. Because the microcontroller cannot measure resistance directly, the change in flex sensor resistance must be converted to a change in a DC voltage. This is accomplished using the voltage divider network shown in Figure 7.5c. For increased flex, the DC voltage will increase. The voltage can be measured using the ATmega164's ADC subsystem.

The flex sensor may be used in applications such as virtual reality data gloves, robotic sensors, biometric sensors, and in science and engineering experiments [Images Company]. One of the coauthors used the circuit provided in Figure 7.5 to help a colleague in zoology monitor the movement of a newt salamander during a scientific experiment.

Example: Ultrasonic sensor

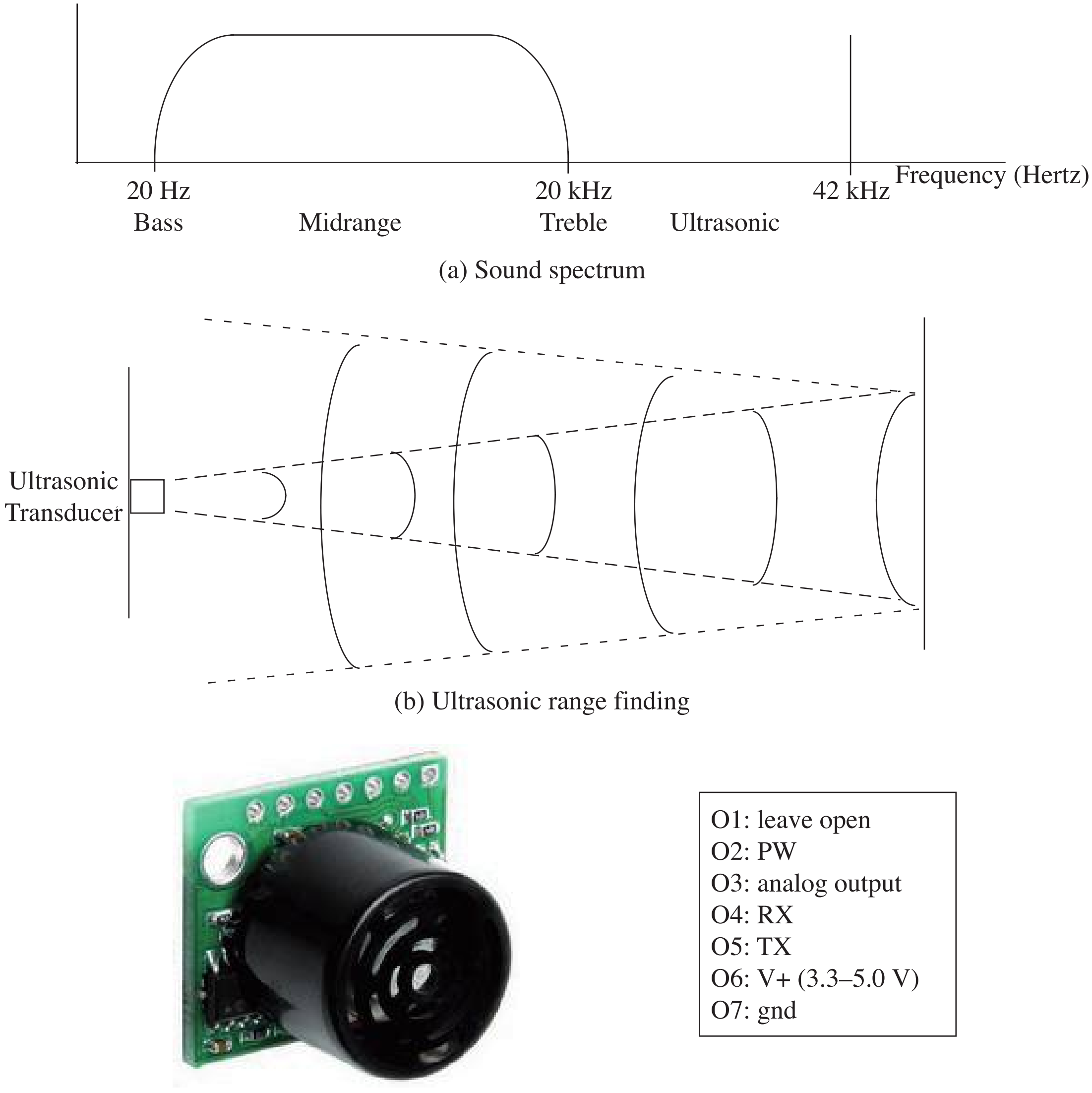
The ultrasonic sensor pictured in Figure 7.6 is an example of an analog based sensor. The sensor is based on the concept of ultrasound or sound waves that are at a frequency above the human range of hearing (20 Hz to 20 kHz). The ultrasonic sensor pictured in Figure 7.6c emits a sound wave at 42 kHz. The sound wave reflects from a solid surface and returns back to the sensor. The amount of time for the sound wave to transit from the surface and back to the sensor may be used to determine the range from the sensor to the wall. Pictured in Figure 7.6c,d is an ultrasonic sensor manufactured by Maxbotix (LV-EZ3). The sensor provides an output that is linearly related to range in three different formats: (a) a serial RS-232 compatible output at 9600 bits per second, (b) a pulse width modulated (PWM) output at a 147 μs/inch duty cycle, and (c) an analog output at a resolution of 10 mV/inch. The sensor is powered from a 2.5–5.5 VDC source (`www.sparkfun.com`).

7.3 OUTPUT DEVICES

As previously mentioned, an external device should not be connected to a microcontroller without first performing careful interface analysis to ensure the voltage, current, and timing requirements of the microcontroller and the external device are met. In this section, we describe interface considerations for a wide variety of external devices. We begin with the interface for a single LED.

7.3.1 LIGHT-EMITTING DIODES

A LED is typically used as a logic indicator to inform the presence of a logic 1 or a logic 0 at a specific pin of a microcontroller. An LED has two leads: the anode or positive lead and the cathode or negative lead. To properly bias an LED, the anode lead must be biased at a level approximately 1.7–2.2 V higher than the cathode lead. This specification is known as the forward voltage (V_f) of the LED. The LED current must also be limited to a safe level known as the forward current (I_f). The diode voltage and current specifications are usually provided by the manufacturer.



(a) Sound spectrum

(b) Ultrasonic range finding

O1: leave open
O2: PW
O3: analog output
O4: RX
O5: TX
O6: V+ (3.3–5.0 V)
O7: gnd

(c) Ultrasonic range finder Maxbotix LV-EZ3
(SparkFun SEN-08501)

(d) Pinout

Figure 7.6: Ultrasonic sensor. (Sensor image used courtesy of SparkFun, Electronics.)

An example of an LED biasing circuit is shown in Figure 7.7a. A logic 1 is provided by the microcontroller to the input of the inverter. The inverter provides a logic 0 at its output, which provides a virtual ground at the cathode of the LED. Therefore, the proper voltage biasing for the LED is provided. The resistor (R) limits the current through the LED. A proper resistor value can be calculated using $R = (V_{DD} - V_{DIODE})/I_{DIODE}$. It is important to note that a 7404 inverter must be used because its capability to safely sink 16 mA of current. Alternately, an NPN transistor such as a 2N2222 (PN2222 or MPQ2222) may be used in place of the inverter as shown in the figure.

7.3.2 SEVEN-SEGMENT LED DISPLAYS

To display numeric data, seven-segment LED displays are available, as shown in Figure 7.7b. Different numerals can be displayed by asserting the proper LED segments. For example, to display the number 5, segments a, c, d, f, and g would be illuminated. Seven-segment displays are available in common cathode (CC) and common anode (CA) configurations. As the CC designation implies, all seven individual LED cathodes on the display are tied together.

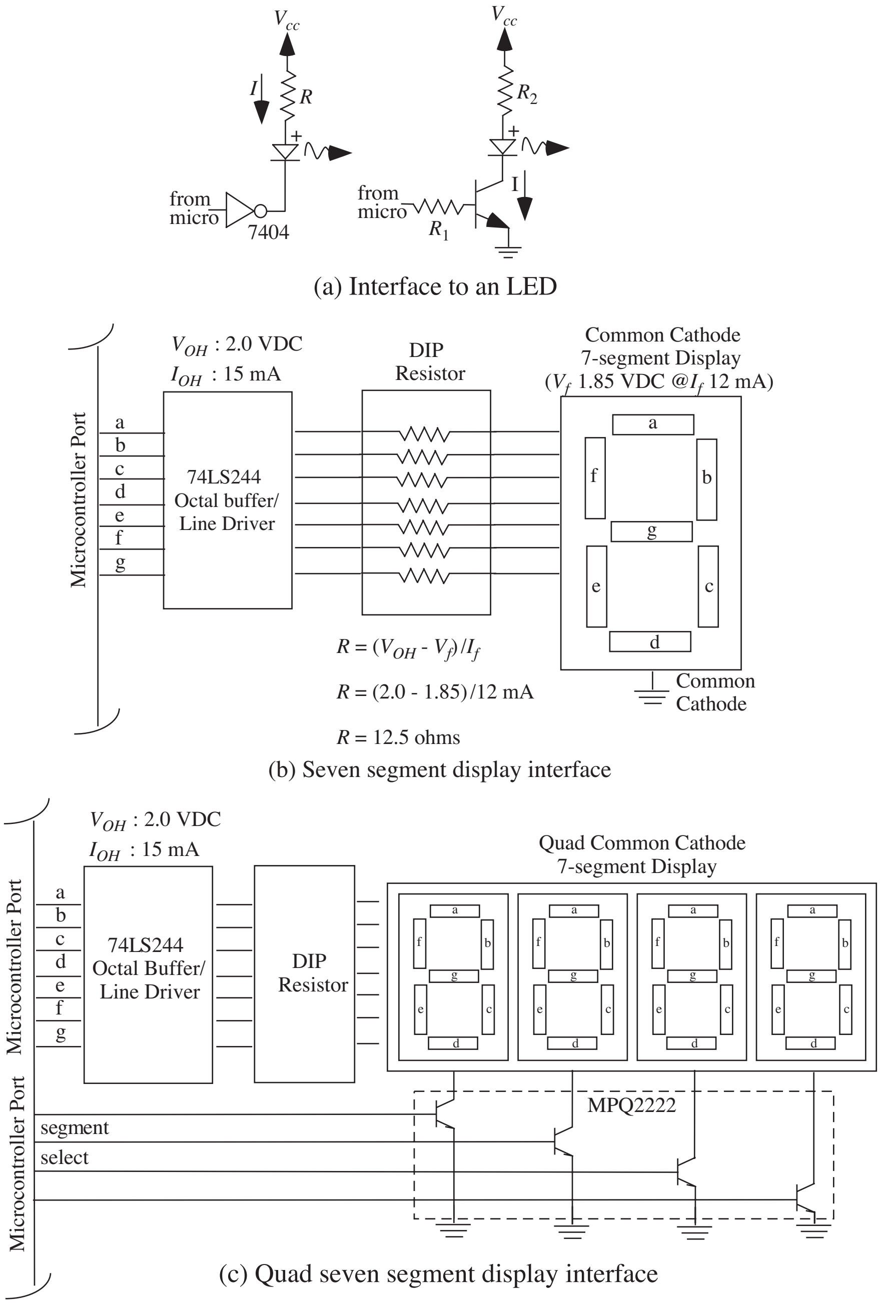
The microcontroller is not capable of driving the LED segments directly. As shown in Figure 7.7b, an interface circuit is required. We use a 74LS244 octal buffer/driver circuit to boost the current available for the LED. The LS244 is capable of providing 15 mA per segment (I_{OH}) at 2.0 VDC (V_{OH}). A limiting resistor is required for each segment to limit the current to a safe value for the LED. Conveniently, resistors are available in DIP packages of eight for this type of application.

Seven-segment displays are available in multicharacter panels. In this case, separate microcontroller ports are not used to provide data to each seven-segment character. Instead, a single port is used to provide character data. A portion of another port is used to sequence through each of the characters as shown in Figure 7.7c. An NPN (for a CC display) transistor is connected to the common cathode connection of each individual character. As the base contact of each transistor is sequentially asserted, the specific character is illuminated. If the microcontroller sequences through the display characters at a rate greater than 30 Hz, the display will have steady illumination.

7.3.3 TRISTATE LED INDICATOR

The tristate LED indicator introduced earlier is shown in Figure 7.8. It is used to provide the status of an entire microcontroller port. The indicator bank consists of eight green and eight red LEDs. When an individual port pin is logic high, the green LED is illuminated. When logic low the red LED is illuminated. If the port pin is at a tristate high-impedance state, no LED is illuminated.

The NPN/PNP transistor pair at the bottom of the figure provides a 2.5 VDC voltage reference for the LEDs. When a specific port pin is logic high (5.0 VDC), the green LED will be forward biased because its anode will be at a higher potential than its cathode. The 47 Ω



(a) Interface to an LED

(b) Seven segment display interface

(c) Quad seven segment display interface

Figure 7.7: LED display devices: (a) interface to an LED, (b) seven-segment display interface, and (c) quad seven-segment display interface.

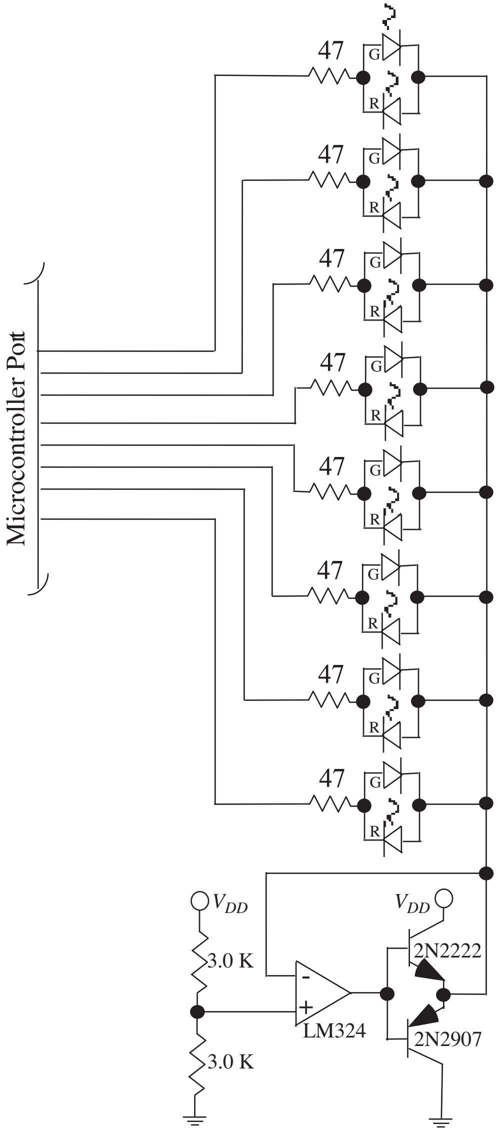


Figure 7.8: Tristate LED display.

resistor limits current to a safe value for the LED. Conversely, when a specific port pin is at a logic low (0 VDC), the red LED will be forward biased and illuminate. For clarity, the red and green LEDs are shown as being separate devices. LEDs are available that have both LEDs in the same device.

7.3.4 DOT MATRIX DISPLAY

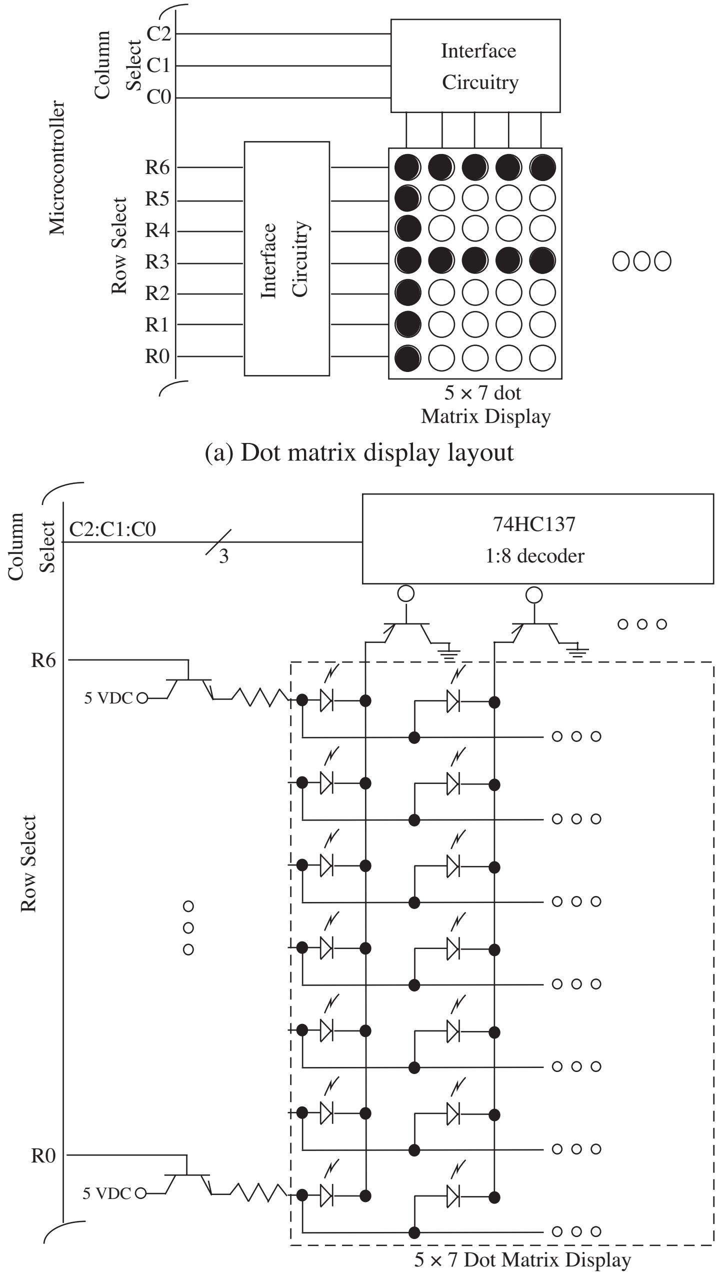
The dot matrix display consists of a large number of LEDs configured in a single package. A typical 5 × 7 LED arrangement is a matrix of five columns of LEDs with seven LEDs per row, as shown in Figure 7.9. Display data for a single matrix column [R6-R0] is provided by the microcontroller. That specific row is then asserted by the microcontroller using the column select lines (C2 C0). The entire display is sequentially built up a column at a time. If the microcontroller sequences through each column fast enough (greater than 30 Hz), the matrix display appears to be stationary to a human viewer.

In Figure 7.9a, we have provided the basic configuration for the dot matrix display for a single-display device. However, this basic idea can be expanded in both dimensions to provide a multicharacter, multiline display. A larger display does not require a significant number of microcontroller pins for the interface. The dot matrix display may be used to display alphanumeric data as well as graphics data. In Figure 7.9b, we have provided additional detail of the interface circuit.

7.3.5 LIQUID CRYSTAL DISPLAY

An LCD is an output device to display text information as shown in Figure 7.10. LCDs come in a wide variety of configurations including multicharacter, multiline format. A 16 × 2 LCD format is common. That is, it has the capability of displaying two lines of 16 characters each. The characters are sent to the LCD via ASCII format a single character at a time. For a parallel-configured LCD, an 8-bit data path and two lines are required between the microcontroller and the LCD. A small microcontroller mounted to the back panel of the LCD translates the ASCII data characters and control signals to properly display the characters. LCDs are configured for either parallel or serial data transmission format. In the example provided, we use a parallel-configured display. In Figure 7.11, we have included the LCD in the Testbench hardware configuration.

Some sample C code is provided below to send data and control signals to an LCD. In this specific example, an AND671GST 1 × 16 character LCD was connected to the Microchip ATmega164 microcontroller [Microchip]. One 8-bit port and two extra control lines are required to connect the microcontroller to the LCD. Note: The initialization sequence for the LCD is specified within the manufacturer's technical data.



(a) Dot matrix display layout

(b) Dot matrix interface details

Figure 7.9: Dot matrix display: (a) dot matrix display and (b) dot matrix interface details.

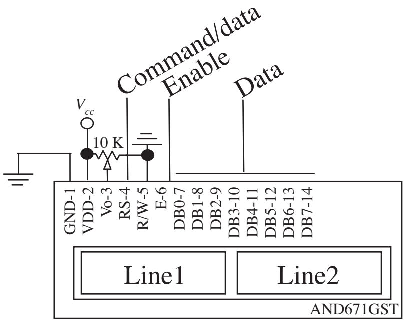


Figure 7.10: LCD display.

```
//*************************************************************
//LCD_Init:initialization for an LCD connected in the following
//manner:
//LCD: AND671GST 1x16 character display
//LCD configured as two 8 character lines in a 1x16 array
//LCD data bus (pin 14-pin7)   MICROCHIP ATmega164: PORTC
//LCD RS (pin 4)   MICROCHIP ATmega164: PORTD[7]
//LCD E  (pin 6)   MICROCHIP ATmega164: PORTD[6]
//*************************************************************

void LCD_Init(void)
{
delay_5ms();
delay_5ms();
delay_5ms();                //output command string to
                            //initialize LCD
putcommand(0x38);           //function set 8-bit
delay_5ms();
putcommand(0x38);           //function set 8-bit
putcommand(0x38);           //function set 8-bit
putcommand(0x38);           //one line, 5x7 char
putcommand(0x0C);           //display on
putcommand(0x01);           //display clear-1.64 ms
putcommand(0x06);           //entry mode set
putcommand(0x00);           //clear display, cursor at home
putcommand(0x00);           //clear display, cursor at home
}
```

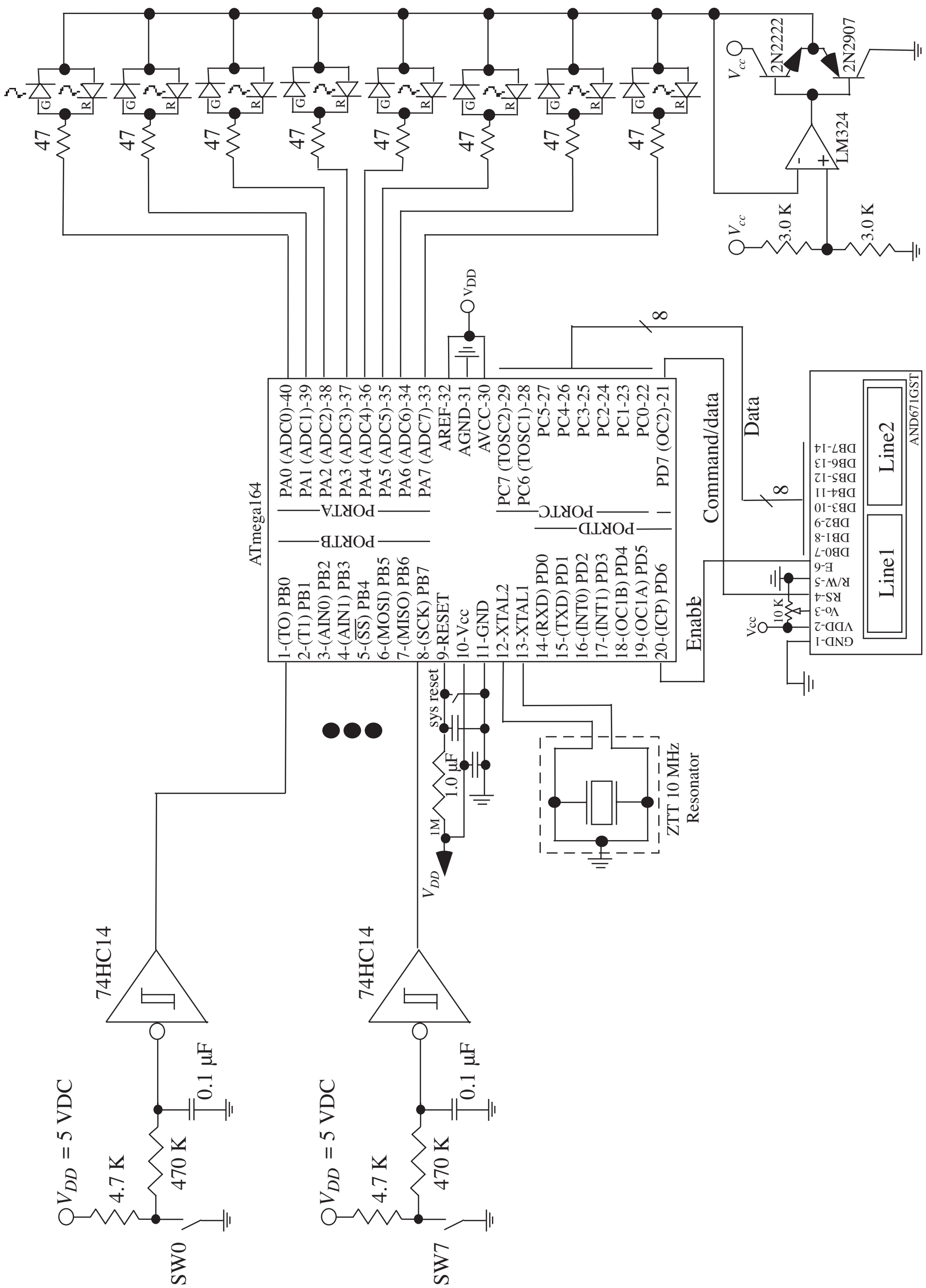
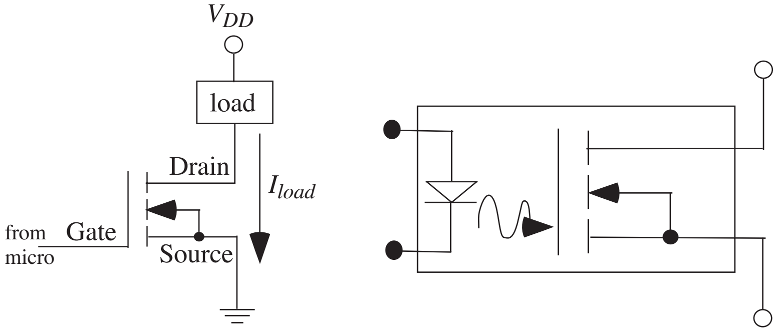


Figure 7.11: Hardware testbench equipped with an LCD.

```
//************************************************************
//putchar:prints specified ASCII character to LCD
//************************************************************

void putchar(unsigned char c)
{
DDRC = 0xff;                    //set PORTC as output
DDRD = DDRD|0xC0;               //make PORTD[7:6] output
PORTC = c;
PORTD = PORTD|0x80;             //RS=1
PORTD = PORTD|0x40;             //E=1
PORTD = PORTD&0xbf;             //E=0
delay_5ms();
}

//************************************************************
//performs specified LCD related command
//************************************************************

void putcommand(unsigned char d)
{
DDRC = 0xff;                    //set PORTC as output
DDRD = DDRD|0xC0;               //make PORTD[7:6] output
PORTD = PORTD&0x7f;             //RS=0
PORTC = d;
PORTD = PORTD|0x40;             //E=1
PORTD = PORTD&0xbf;             //E=0
delay_5ms();
}

//**************************************************************
```

7.3.6 HIGH-POWER DC DEVICES

A number of direct current devices may be controlled with an electronic switching device such as a MOSFET. Specifically, an N-channel enhancement MOSFET (metal oxide semiconductor field effect transistor) may be used to switch a high-current load on and off (such as a motor) using a low-current control signal from a microcontroller as shown in Figure 7.12a. The low-current control signal from the microcontroller is connected to the gate of the MOSFET. The



(a) N-channel enhance MOSFET

(b) Solid-state relay with optical interface

Figure 7.12: MOSFET circuits: (a) N-channel enhance MOSFET and (b) solid-state relay (SSR) with optical interface.

MOSFET switches the high-current load on and off consistent with the control signal. The high-current load is connected between the load supply and the MOSFET drain. It is important to note that the load supply voltage and the microcontroller supply voltage do not have to be at the same value. When the control signal on the MOSFET gate is logic high, the load current flows from drain to source. When the control signal applied to the gate is logic low, no load current flows. Thus, the high-power load is turned on and off by the low-power control signal from the microcontroller.

Often the MOSFET is used to control a high-power motor load. A motor is a notorious source of noise. To isolate the microcontroller from the motor noise, an optical isolator may be used as an interface as shown in Figure 7.12b. The link between the control signal from the microcontroller to the high-power load is via an optical link contained within an SSR. The SSR is properly biased using techniques previously discussed.

7.4 DC MOTOR SPEED AND DIRECTION CONTROL

Often, a microcontroller is used to control a high-power motor load. To properly interface the motor to the microcontroller, we must be familiar with the different types of motor technologies. Motor types are illustrated in Figure 7.13.

- **DC motor:** A DC motor has a positive and negative terminal. When a DC power supply of suitable current rating is applied to the motor, it will rotate. If the polarity of the supply is switched with reference to the motor terminals, the motor will rotate in the opposite direction. The speed of the motor is roughly proportional to the applied voltage up to the rated voltage of the motor.

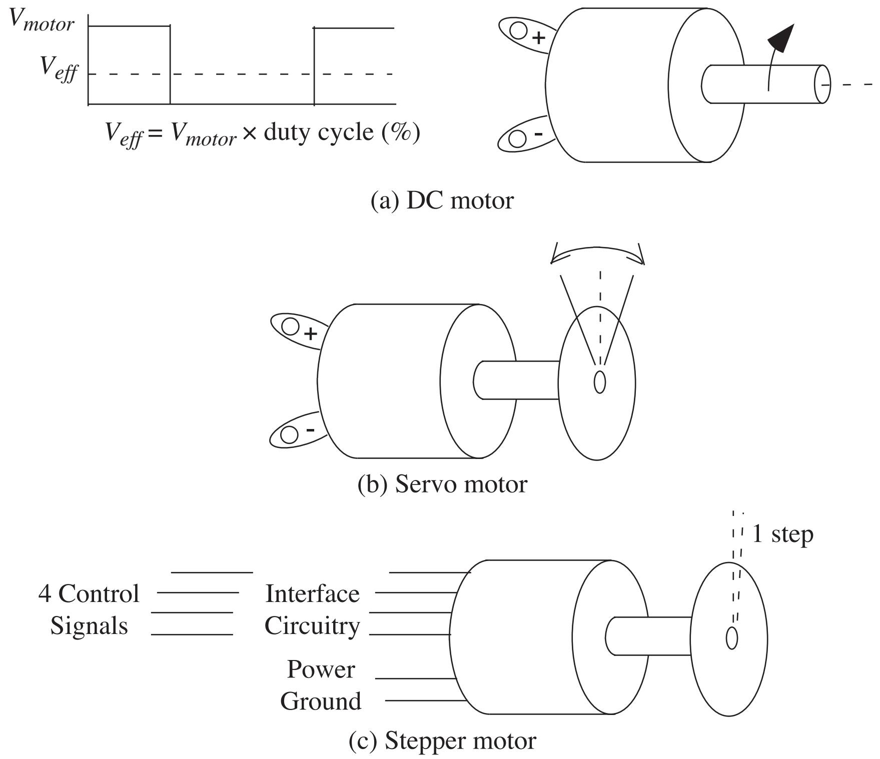


Figure 7.13: Motor types: (a) DC, (b) servo, and (c) stepper.

- **Servo motor:** A servo motor provides a precision angular rotation for an applied PWM duty cycle. As the duty cycle of the applied signal is varied, the angular displacement of the motor also varies. This type of motor is used to change mechanical positions such as the steering angle of a wheel.

- **Stepper motor:** A stepper motor, as its name implies, provides an incremental step change in rotation (typically 2.5° per step) for a step change in control signal sequence. The motor is typically controlled by a two- or four-wire interface. For the four-wire stepper motor, the microcontroller provides a 4-bit control sequence to rotate the motor clockwise. To turn the motor counterclockwise, the control sequence is reversed. The low-power control signals are interfaced to the motor via MOSFETs or power transistors to provide for the proper voltage and current requirements of the pulse sequence.

7.4.1 H-BRIDGE DIRECTION CONTROL

For a DC motor to operate in both the clockwise and counter clockwise directions, the polarity of the DC voltage supplied must be changed. To operate the motor in the forward direction, the positive battery terminal must be connected to the positive motor terminal while the negative battery terminal must be provided to the negative motor terminal. To reverse the motor direction the motor supply polarity must be reversed. An H-bridge is a circuit employed to perform this polarity switch. Low power H-bridges (500 mA) come in a convenient dual in line package (e.g., 754110). For higher power motors, a H-bridge may be constructed from discrete components. Figure 7.14a provides an interface circuit to drive a DC motor in one direction. A pulse width modulated signal (PWM) may be applied to the base of the transistor to control motor speed. Figure 7.14b shows an H-bridge circuit for bi-directional motor control. If PWM signals are used to drive the base of the transistors (from microcontroller pins PD4 and PD5), both motor speed and direction may be controlled by the circuit. The transistors used in the circuit must have a current rating sufficient to handle the current requirements of the motor during start and stall conditions. To rotate the motor in one direction, a PWM signal is provided to PD4 and pin PD5 is provided with a logic low signal. To rotate the motor in the opposite direction, pin PD4 is provided with a logic low signal while PD5 is provided with the PWM signal.

7.4.2 SERVO MOTOR INTERFACE

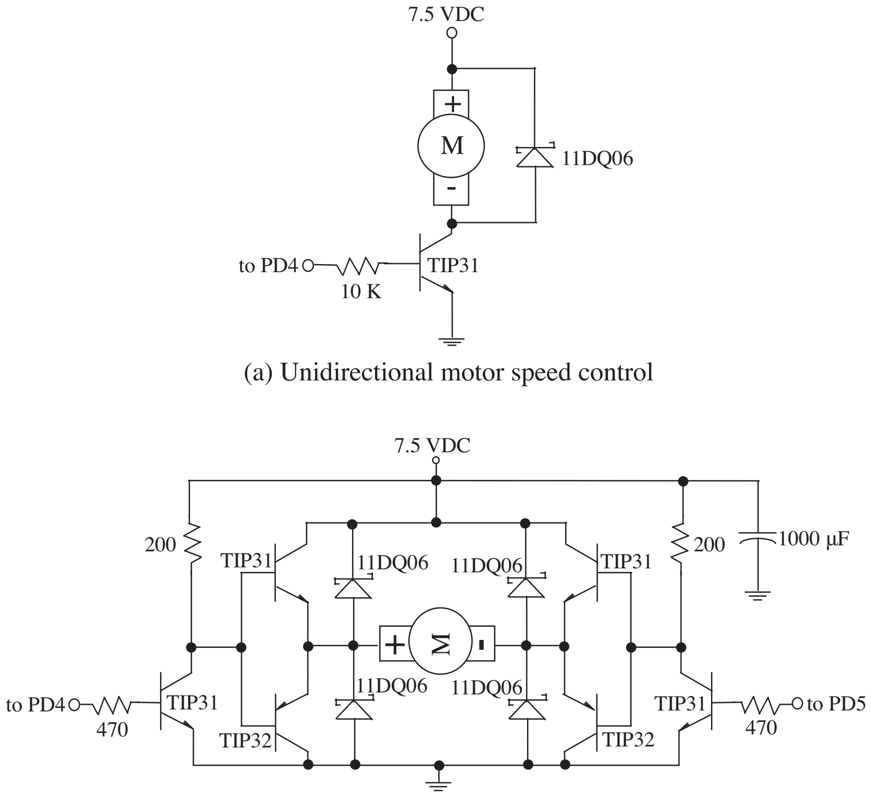
The servo motor is used for a precise angular displacement. The displacement is related to the duty cycle of the applied control signal. A servo control circuit and supporting software was presented in Chapter 5.

7.4.3 STEPPER MOTOR CONTROL

Stepper motors are used to provide a discrete angular displacement in response to a control signal step. There are a wide variety of stepper motors including bipolar and unipolar types with different configurations of motor coil wiring. Due to space limitations we only discuss the unipolar, 5-wire stepper motor. The internal coil configuration for this motor is provided in Figure 7.15b.

Often a wiring diagram is not available for the stepper motor. Based on the wiring configuration (Figure 7.15b), one can determine the common line for both coils. It will have a resistance that is one-half of all of the other coils. Once the common connection is found, one can connect the stepper motor into the interface circuit. By changing the other connections, one can determine the correct connections for the step sequence.

To rotate the motor either clockwise or counter clockwise, a specific step sequence must be sent to the motor control wires as shown in Figure 7.15c. As shown in Figure 7.15c, the control sequence is transmitted by four pins on the microcontroller. In this example we use PORTD[7:5].



(a) Unidirectional motor speed control

(b) Bidirectional motor speed control

Figure 7.14: (a) Uni-directional motor control circuit, and (b) bi-directional H-bridge control circuit.

The microcontroller does not have sufficient capability to drive the motor directly. Therefore, an interface circuit is required, as shown in Figure 7.15c. For a unipolar stepper motor, we employ a TIP130 power Darlington transistor to drive each coil of the stepper motor. The speed of motor rotation is determined by how fast the control sequence is completed. The TIP 30 must be powered by a supply that has sufficient capability for the stepper motor coils.

Example: An ATmega324 has been connected to a JRP 42BYG016 unipolar, 1.8° per step, 12 VDC at 160 mA stepper motor. The interface circuit is shown in Figure 7.15. PORTD pins 7 to 4 are used to provide the step sequence. A one second delay is used between the steps to control motor speed. Pushbutton switches are used on PORTB[1:0] to assert CW and CCW stepper motion. An interface circuit consisting of four TIP130 transistors are used between the

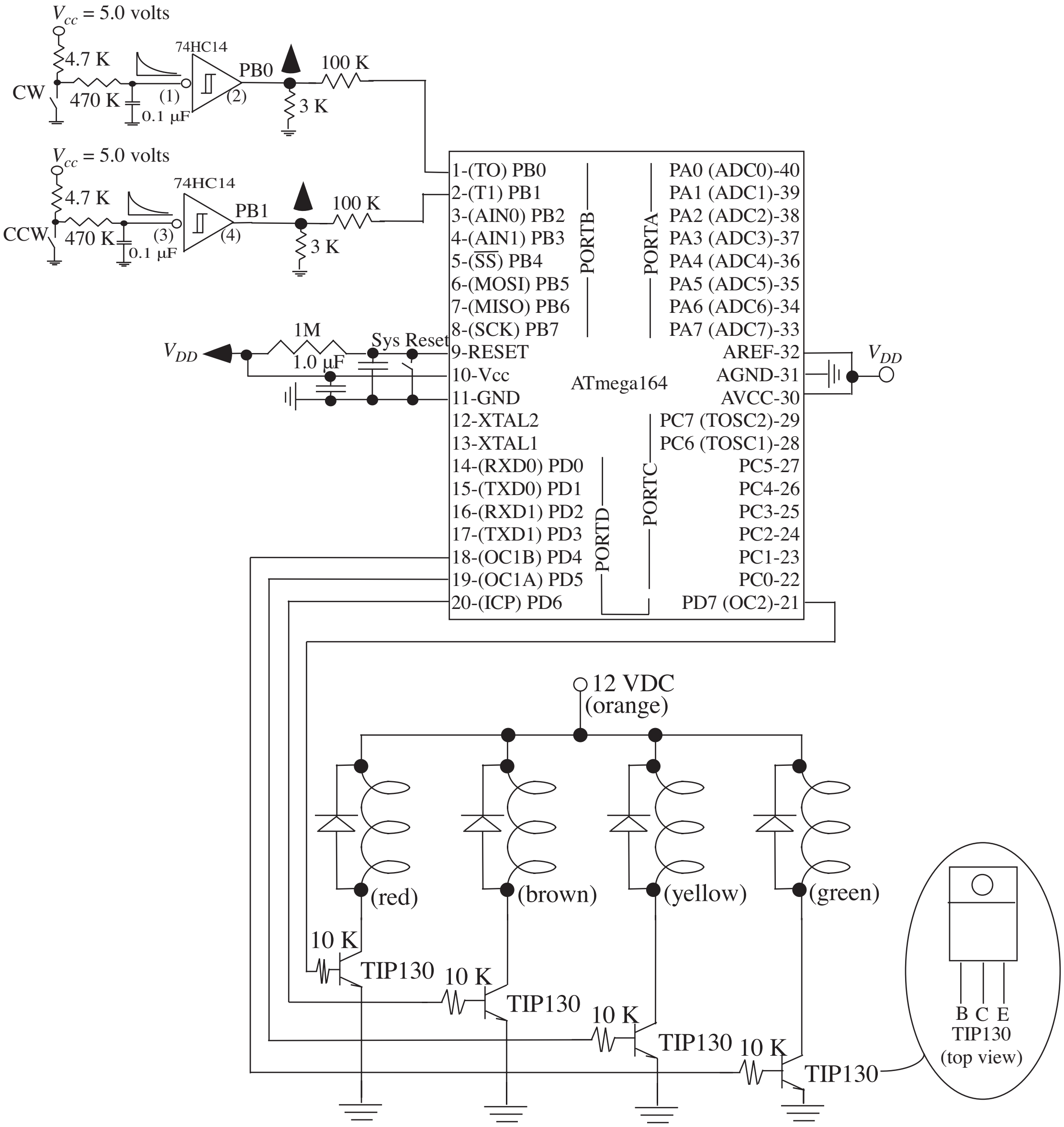


Figure 7.15: Unipolar stepper motor control circuit.

microcontroller and the stepper motor to boost the voltage and current of the control signals. Code to provide the step sequence is shown below.

Provided below is a basic function to rotate the stepper motor in the forward or reverse direction.

```
//*************************************************************
//target controller: MICROCHIP ATmega324
//
//MICROCHIP AVR ATmega324PV Controller Pin Assignments
//Chip Port Function I/O Source/Dest Asserted Notes
//PORTB:
//Pin 1 PB0 to active high RC debounced switch - CW
//Pin 2 PB1 to active high RC debounced switch - CCW
//Pin 9 Reset - 1M resistor to Vcc, tact switch to ground,
//1.0 uF to ground
//Pin 10 Vcc - 1.0 uF to ground
//Pin 11 Gnd
//Pin 12 ZTT-10.00MT ceramic resonator connection
//Pin 13 ZTT-10.00MT ceramic resonator connection
//Pin 18 PD4 - to servo control input
//Pin 30 AVcc to Vcc
//Pin 31 AGnd to Ground
//Pin 32 ARef to Vcc
//*************************************************************

//include files
//MICROCHIP register definitions for ATmega324
#include<iom324pv.h>
#include<macros.h>
                                  //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:19


//function prototypes*****************************************
void initialize_ports(void);      //initialize ports
void power_on_reset(void);        //return sys to startup state
void read_new_input(void);        //read input change on PORTB
void init_timer0_ovf_interrupt(void); //init timer0 overflow
void timer0_interrupt_isr(void);
void delay(unsigned int);
```

```
//main program*************************************************
//The main program checks PORTB for user input activity. If new activity
//is found, the program responds.

//global variables
unsigned char   old_PORTB = 0x08; //present value of PORTB
unsigned char   new_PORTB;        //new values of PORTB
unsigned int    input_delay;      //delay counter - increment via Timer0
                                  //overflow interrupt

void main(void)
{
initialize_ports();               //return LED configuration to default
init_timer0_ovf_interrupt();      //used to initialize timer0 overflow

while(1)
  {
  _StackCheck();                  //check for stack overflow
  read_new_input();               //read input status changes on PORTB
  }
}//end main

//Function definitions

//**************************************************************
//initialize_ports: provides initial configuration for I/O ports
//**************************************************************

void initialize_ports(void)
{
//PORTA
PORTA DDRA=0xff;                  //PORTA[7:0] output
PORTA=0x00;                       //turn off pull ups

//PORTB
DDRB=0xfc;                        //PORTB[7-2] out, PORTB[1:0] in
PORTB=0x00;                       //disable pull-up resistors
```

```
//PORTC
DDRC=0xff;                        //set PORTC[7-0] as output
PORTC=0x00;                       //init low

//PORTD
DDRD=0xff;                        //set PORTD[7-0] as output
PORTD=0x00;                       //initialize low
}

//***************************************************************
//read_new_input: function polls PORTB for a change in status.
//If status change has occurred, appropriate function for status
//change is called.
//  - Pin 1 PB0 to active high RC  debounced switch - CW
//  - Pin 2 PB1 to active high RC debounced switch - CCW
//***************************************************************

void read_new_input(void)
{
new_PORTB = (PINB & 0x03);
if(new_PORTB != old_PORTB)
  {
  switch(new_PORTB){              //process change in PORTB input

    case 0x01:                    //CW
      while((PINB & 0x03)==0x01)
        {
        PORTD = 0x80;
        delay(15);                //delay 1s
        PORTD = 0x00;
        delay(1);                 //delay 65 ms

        PORTD = 0x40;
        delay(15);
        PORTD = 0x00;
        delay(1);

        PORTD = 0x20;
```

```
          delay(15);
          PORTD = 0x00;
          delay(1);

          PORTD = 0x10;
          delay(15);
          PORTD = 0x00;
          delay(1);
          }
          break;

     case
       0x02:                          //CCW
         while((PINB & 0x03)==0x02)
         {
         PORTD = 0x10;
         delay(15);
         PORTD = 0x00;
         delay(1);

         PORTD = 0x20;
         delay(15);
         PORTD = 0x00;
         delay(1);

         PORTD = 0x40;
         delay(15);
         PORTD = 0x00;
         delay(1);

         PORTD = 0x80;
         delay(15);
         PORTD = 0x00;
         delay(1);
   }
         break;

        default:;                     //all other cases
        }                             //end switch(new_PORTB)
```

```
  }                                //end if new_PORTB
  old_PORTB=new_PORTB;             //update PORTB
}

//*************************************************************
//int_timer0_ovf_interrupt(): The Timer0 overflow interrupt
//is being employed as a time base for a master timer for this
//project. The internal oscillator of 8 MHz is divided
//internally by 8 to provide a 1 MHz time base and is divided
//by 256.  The 8-bit Timer0 register (TCNT0) overflows every
//256 counts or every 65.5 ms.
//*************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04;                      //divide timer0 timebase
                                    //by 256, overflow occurs
                                    //every 65.5ms
TIMSK0 = 0x01;                      //en timer0 overflow int
asm("SEI");                         //enable global interrupt
}

//*************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing
//the corresponding interrupt handling vector.
//*************************************************************

void timer0_interrupt_isr(void)
{
input_delay++;                      //input delay processing
}

//*************************************************************
//void delay(unsigned int number_of_65_5ms_interrupts) this
//generic delay function provides the specified delay as the
//number of 65.5 ms "clock ticks" from the Timer0 interrupt.
//Note: this function is only valid when using a 1 MHz crystal
//or ceramic resonator.
```

```
//*************************************************************

void delay(unsigned int number_of_65_5ms_interrupts)
{
TCNT0 = 0x00;                      //reset timer0
input_delay = 0;
while(input_delay <= number_of_65_5ms_interrupts)
  {
  ;
  }
}

//*************************************************************
```

7.4.4 AC DEVICES

In a similar manner, a high-power AC load may be switched on and off using a low-power control signal from the microcontroller. In this case, an SSR is used as the switching device. SSRs are available to switch a high-power DC or AC load (Crydom). For example, the Crydom 558-CX240D5R is a printed circuit board mounted, air-cooled, single-pole, single-throw (SPST), NO SSR. It requires a DC control voltage of 3–15 VDC at 15 mA. However, this small microcontroller-compatible DC control signal is used to switch 12–280 VAC loads rated from 0.06–5 A (Crydom).

To vary the direction of an AC motor, you must use a bidirectional AC motor. A bidirectional motor is equipped with three terminals: common, clockwise, and counterclockwise. To turn the motor clockwise, an AC source is applied to the common and clockwise connections. In like manner, to turn the motor counterclockwise, an AC source is applied to the common and counterclockwise connections. This may be accomplished using two of the Crydom SSRs.

PowerSwitch manufactures an easy-to-use AC interface, the PowerSwitch Tail II. The device consists of a control module with attached AC connections rated at 120 VAC, 15 A. The device to be controlled is simply plugged inline with the PowerSwitch Tail II. A digital control signal from the microcontroller serves as the on/off control signal for the controlled AC device. The controlled signal is connected to the PowerSwitch Tail II via a terminal block connection. The PowerSwitch II is available as either normally closed (NC) or normally open (NO) (www.powerswitchtail.com).

Example: PowerSwitch Tail II. In this example, we use an IR sensor to detect someone's presence. If the IR sensor's output reaches a predetermined threshold level, an AC desk lamp is illuminated, as shown in Figure 7.16.

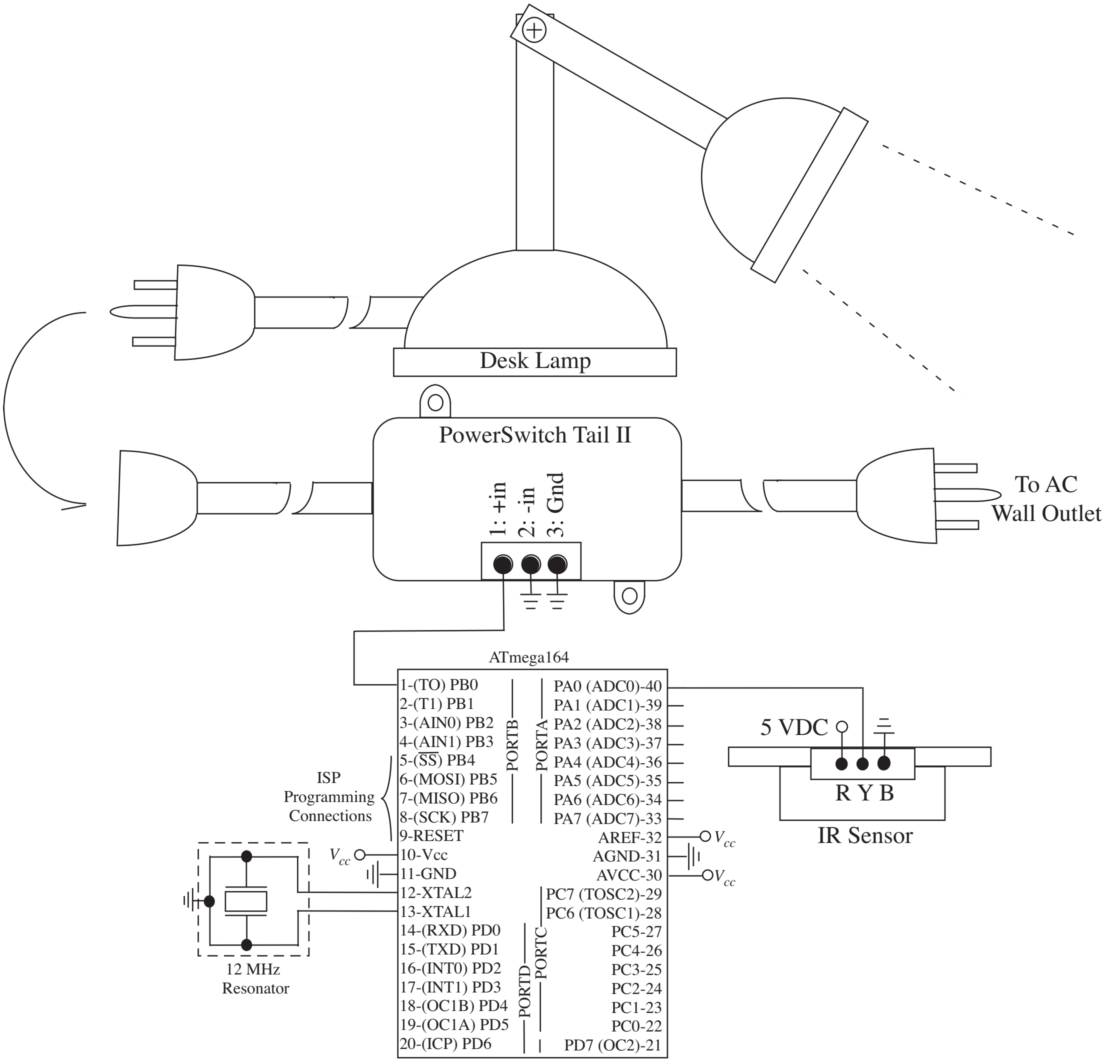


Figure 7.16: PowerSwitch Tail II.

```
//****************************************************************
//file name: switchtail2.c
//****************************************************************

//include files***************************************************

//MICROCHIP register definitions for ATmega164
#include <iom164pv.h>

//function prototypes*********************************************
void delay(unsigned int number_of_26_2ms_interrupts);
void init_timer0_ovf_interrupt(void);
void initialize_ports(void);
void InitADC( void);
unsigned int ReadADC(unsigned char channel);

unsigned int   IR_int;
unsigned int   IR_threshold = 0x00ff;
unsigned int   delay_timer = 0;

int main(void)
{
initialize_ports();                   //initialize ports
InitADC();                            //Initialize ADC
init_timer0_ovf_interrupt();          //init Timer0 for delay
PORTD = PORTD | 0x80;                 //toggle LED PORTD[7]

while(1)
  {
  IR_int = ReadADC(0x00);             //read IR sensor ADC[0]
  if(IR_int >=  IR_threshold)
    PORTB = PORTB | 0x01;             //turn lamp on
  delay(382);                         //delay 10s
  PORTB = PORTB & 0xfe;               //turn lamp off
  }
}

//****************************************************************
//function definitions********************************************
```

```
//***************************************************************
//initialize_ports: provides initial configuration for I/O ports
//***************************************************************

void initialize_ports(void)
{
DDRA =0xfe;  //set PORTA as output, PORTA[0] input
PORTA=0x00;  //initialize low

DDRB =0x0f;  //PORTB[7:4] as input, set PORTB[3:0] as output
PORTB=0x00;  //disable PORTB pull-up resistors

DDRC =0xff; //set PORTC as output
PORTC=0x00; //initialize low

DDRD =0xff; //set PORTD as output
PORTD=0x00; //initialize low
}

//*************************************************************
//InitADC: initialize analog-to-digital converter
//*************************************************************
void InitADC( void)
{
ADMUX = 0;                      //Select channel 0
ADCSRA = 0xC3;                  //Enable ADC & start 1st
                                //dummy conversion
                                //Set ADC module prescalar
                                //to 8 critical for
                                //accurate ADC results
while (!(ADCSRA & 0x10));       //Check if conversion ready
ADCSRA |= 0x10;                 //Clear conv rdy flag -
                                //set the bit
}

//*************************************************************
//ReadADC: read analog voltage from analog-to-digital converter -
//the desired channel for conversion is passed in as an unsigned
```

```
//character variable. The result is returned as a left justified,
//10 bit binary result. The ADC prescalar must be set to 8 to
//slow down the ADC clock at higher external clock frequencies
//(10 MHz) to obtain accurate results.
//***************************************************************
unsigned int ReadADC(unsigned char channel)
{
unsigned int binary_weighted_voltage, binary_weighted_voltage_low;
unsigned int binary_weighted_voltage_high; //weighted binary
                                 //voltage
ADMUX = channel;                 //Select channel
ADCSRA |= 0x43;                  //Start conversion
                                 //Set ADC module prescalar
                                 //to 8 critical for
                                 //accurate ADC results
while (!(ADCSRA & 0x10));        //Check if conversion ready
ADCSRA |= 0x10;                  //Clear Conv rdy flag - set
                                 //the bit
binary_weighted_voltage_low = ADCL; //Read 8 low bits first
                                 //(important)
                                 //Read 2 high bits,
                                 //multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage = binary_weighted_voltage_low |
                          binary_weighted_voltage_high;
return binary_weighted_voltage;  //ADCH:ADCL
}

//*****************************************************************
//delay(unsigned int num_of_26_2_ms_interrupts): this generic
//delay function provides the specified delay as the number of
//26.2 ms "clock ticks" from the Timer0 interrupt.
//
//Note: this function is only valid when using a 2.5 MHz
//time base.  e.g., 20 MHz ceramic resonator divided by 8
//*****************************************************************

void delay(unsigned int number_of_26_2ms_interrupts)
{
```

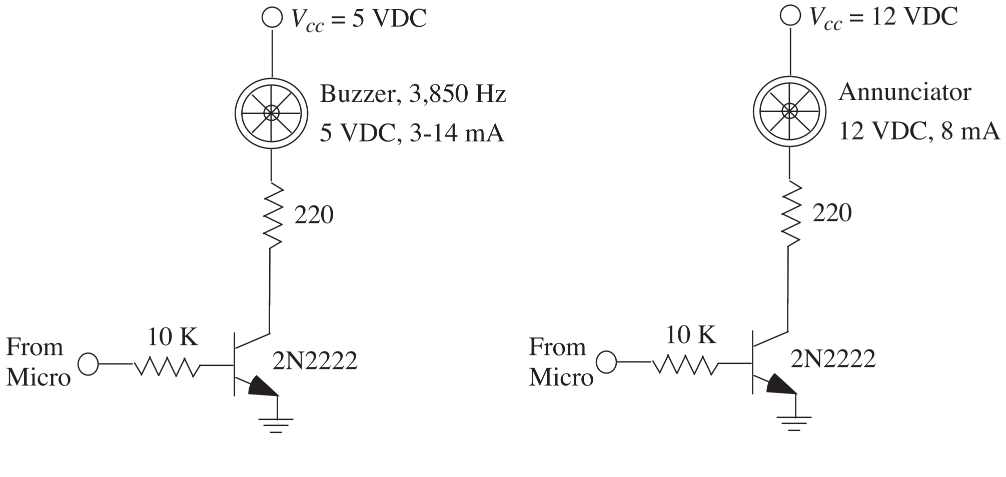
```
TCNT0 = 0x00;                         //reset delay_timer
delay_timer = 0;
while(delay_timer <= number_of_26_2ms_interrupts)
  {
  ;
  }
}

//***************************************************************
//int_timer0_ovf_interrupt(): The Timer0 overflow interrupt is
//being employed as a time base for a master timer for this
//project. The internal time base is set to operate at 2.5 MHz and
//then is divided by 256.  The 8-bit Timer0 register (TCNT0)
//overflows every 256 counts or every 26.2 ms.
//***************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04; //divide timer0 timebase by 256, overflow occurs
               //every 26.2ms
TIMSK0 = 0x01; //enable timer0 overflow interrupt
asm("SEI");    //enable global interrupt
}

//***************************************************************
//void timer0_interrupt_isr(void)
//***************************************************************

void timer0_interrupt_isr(void)
{
delay_timer++;                            //increment timer
}

//***************************************************************
```

7.5 INTERFACING TO MISCELLANEOUS DC DEVICES

In this section, we present a potpourri of interface circuits to connect a microcontroller to a wide variety of DC peripheral devices.

7.5.1 SONALERTS, BEEPERS, BUZZERS

In Figure 7.17, we show several circuits used to interface a microcontroller to a buzzer, a beeper, or other types of annunciator devices such as a sonalert. It is important that the interface transistor and the supply voltage are matched to the requirements of the sound producing device.



(a) 5 VDC buzzer interface

(b) 12 VDC annunciator

Figure 7.17: Sonalert, beepers, and buzzers.

7.5.2 VIBRATING MOTOR

A vibrating motor is often used to gain one's attention as in a cell phone. These motors are typically rated at 3 VDC and a high current. The interface circuit shown in Figure 7.18 is used to drive the low voltage motor. The transistor interface should have a current rating at least twice that of the stall current of the motor.

7.5.3 DC FAN

The interface circuit shown in Figure 7.19 may also be used to control a DC fan. As before, a reverse-biased diode is placed across the DC fan motor.

7.5.4 BILGE PUMP

A bilge pump is a pump specifically designed to remove water from the inside of a boat. The pumps are powered from a 12 VDC source and have typical flow rates from 360 to over 3,500 gallons per minute. (They range in price from U.S. $20 to U.S. $80 (`www.shorelinemarinedevelopment.com`)). An interface circuit to control a bilge pump from MSP432 is shown in Figure 7.20. The interface circuit consists of a 470 ohm resistor, a power NPN Darlington

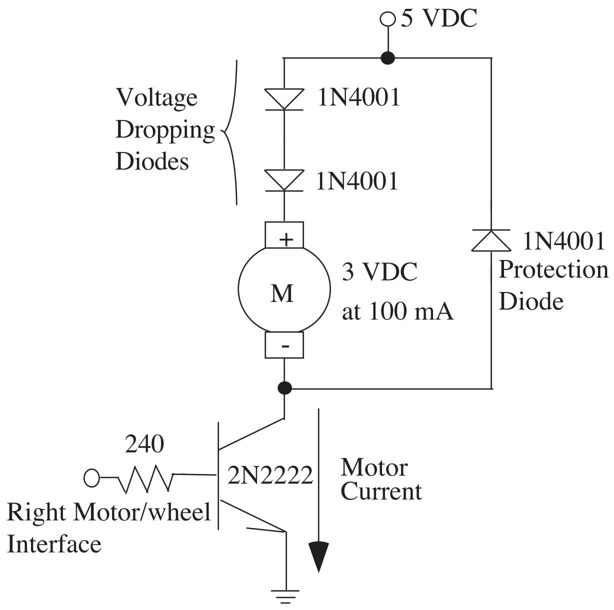


Figure 7.18: Motor driver circuit for low voltage motor.

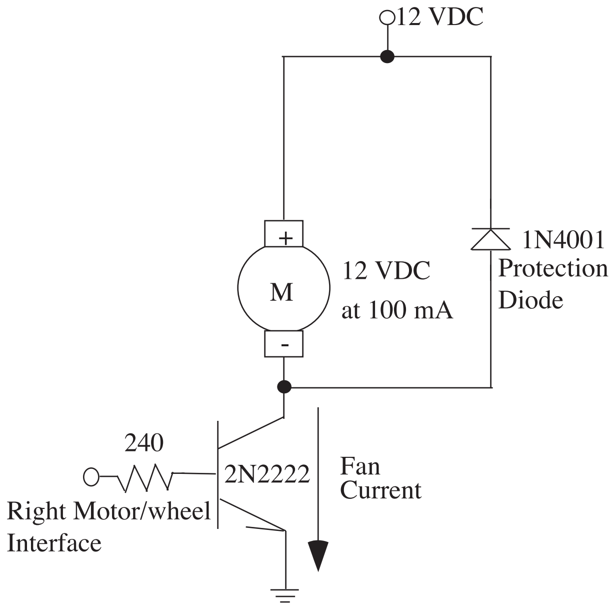
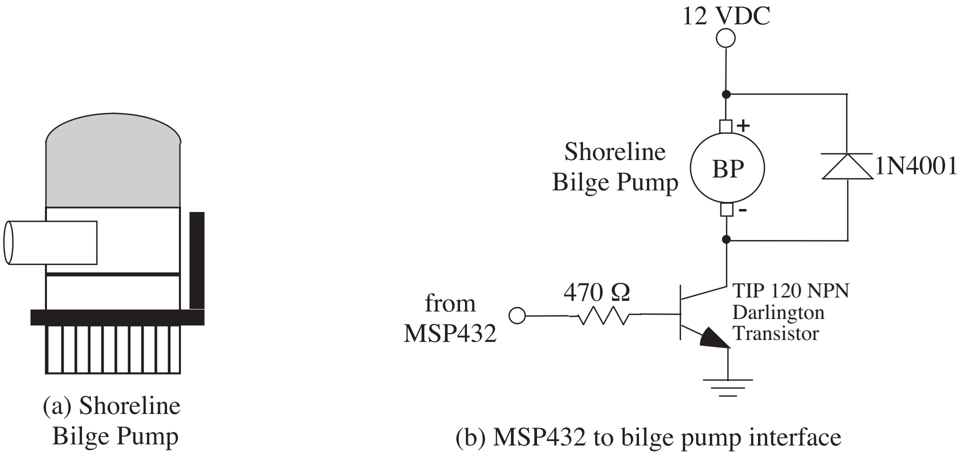


Figure 7.19: Motor driver circuit for 12 VDC fan.



(a) Shoreline Bilge Pump

(b) MSP432 to bilge pump interface

Figure 7.20: Bilge pump interface.

transistor (TIP 120) and a 1N4001 diode. The 12 VDC supply should have sufficient current capability to supply the needs of the bilge pump.

7.6 SUMMARY

In this chapter, we have discussed the voltage and current operating parameters for the Microchip HC CMOS-type microcontroller. We discussed how this information may be applied to properly design an interface for common I/O circuits. It must be emphasized that a properly designed interface allows the microcontroller to operate properly within its parameter envelope. If, because of a poor interface design, a microcontroller is used outside its prescribed operating parameter values, spurious and incorrect logic values will result. We provided interface information for a wide range of I/O devices. We also discussed the concept of interfacing a motor to a microcontroller using PWM techniques coupled with high-power MOSFET or SSR switching devices.

7.7 REFERENCES AND FURTHER READING

Atmel 8-bit AVR Microcontroller with 16/32/64/128K Bytes In-System Programmable Flash, ATmega164A, ATmega164PA, ATmega324A, ATmega324PA, ATmega644A, ATmega644PA, ATmega1284, ATmega1284P, 8272C-AVR-06/11, data sheet: 8272C-AVR-06/11, Atmel, San Jose, CA, 2015. 220

Barrett, S. F. and Pack, D. J. *Embedded Systems Design with the 68HC12 and HCS12*. Prentice Hall, Upper Saddle River, NJ, 2004.

Barrett, S. F. and Pack, D. J. *Microcontrollers Fundamentals for Engineers and Scientists*. Morgan & Claypool, San Rafael, CA, 2006. DOI: 10.2200/s00025ed1v01y200605dcs001. 205

Barrett, S. F. and Pack, D. J. *68HC12 Microcontroller: Theory and Applications*. Prentice Hall, Upper Saddle River, NJ, 2002.

Crydom: PCB Mount SSRs, San Diego, CA, 2019. www.crydom.com

Resistive Flex Sensors, Spectra Symbol, www.spectrasymbol.com, 2019. 215

Stegmann: Absolute Encoder, Dayton, OH, 2019. www.stegmann.com 214

7.8 CHAPTER PROBLEMS

1. What will happen if a microcontroller is used outside its prescribed operating envelope?
2. Discuss the difference between the terms *sink* and *source* as related to current loading of a microcontroller.
3. Can an LED with a series limiting resistor be directly driven by the Microchip microcontroller? Explain.
4. In your own words, provide a brief description of each of the microcontroller electrical parameters.
5. What is switch bounce? Describe two techniques to minimize switch bounce.
6. Describe a method of debouncing a keypad.
7. What is the difference between an incremental encoder and an absolute encoder? Describe applications for each type.
8. What must be the current rating of the 2N2222 and 2N2907 transistors used in the tristate LED circuit? Support your answer.
9. Draw the circuit for a six-character, seven-segment display. Fully specify all components. Write a program to display "ATmega164."
10. Repeat the question above for a dot matrix display.
11. Repeat the question above for an LCD display.

CHAPTER 8

Embedded Systems Design

Objectives: After reading this chapter, the reader should be able to:

- define an embedded system;
- list factors governing the design of an embedded system;
- provide a step-by-step approach to embedded system design;
- explain design tools and practices of embedded systems design;
- apply embedded system design practices to an Microchip microcontroller-based embedded system; and
- provide detailed designs for systems controlled by an Microchip microcontroller, including hardware layouts, interfaces, structure and UML activity diagrams, and coded algorithms.

This chapter provides a step-by-step methodical approach to designing embedded systems. We begin with a definition of an embedded system. We then explore the process of how to successfully (and with low stress) develop an embedded system prototype that meets established requirements. We conclude the chapter with several extended examples.

8.1 WHAT IS AN EMBEDDED SYSTEM?

An embedded system contains a microcontroller to perform its job of processing system inputs and generating system outputs. The link between system inputs and outputs is provided by a coded algorithm stored within the processor's resident memory. What makes embedded systems design so interesting and challenging is the design must also take into account the proper electrical interface for the input and output devices, limited on-chip resources, human interface concepts, the operating environment of the system, cost analysis, related standards, and manufacturing aspects [Anderson, 2008]. Through careful application of this material, you will be able to design and prototype embedded systems, based on the Microchip microcontroller fundamentals, discussed throughout the book.

8.2 EMBEDDED SYSTEM DESIGN PROCESS

In this section, we show a step-by-step approach to develop the first prototype of an embedded system that will meet established requirements. There are many formal design processes that we could study. We concentrate on the steps that are common to most. We purposefully avoid formal terminology of a specific approach, and instead, concentrate on the activities that are accomplished as a system prototype is developed. The design process we describe is illustrated in Figure 8.1 using a unified modeling language (UML) activity diagram. We discuss the UML activity diagrams later in the chapter.

8.2.1 PROJECT DESCRIPTION

The goal of the project description step is to determine what the system is ultimately supposed to do. To achieve this step you must thoroughly investigate the functions of the system. Questions to raise and answer during this step include but are not limited to the following.

- What is the system supposed to do?
- Where will it be operating and under what conditions?
- Are there any restrictions placed on the system design?

To answer these questions, a designer must interact with the client to ensure clear agreement on what is to be done. If you are completing this project for yourself, you must still carefully and thoughtfully complete this step. The establishment of clear, definable system requirements may require considerable interaction between the designer and the client. It is essential that both parties agree on system requirements before proceeding further in the design process. The final result of this step is a detailed listing of system requirements and related specifications.

8.2.2 BACKGROUND RESEARCH

Once a detailed list of requirements has been established, the next step is to perform background research related to the design. In this step, the designer will ensure they understand all requirements and features required by the project. This will again involve interaction between the designer and the client. The designer will also investigate applicable codes, guidelines, protocols, and standards related to the project. This is also a good time to start thinking about the interface between different portions of the project, particularly the input and output devices peripherally connected to the microcontroller. The ultimate objective of this step is to have a thorough understanding of the project requirements, related project aspects, and any interface challenges within the project.

8.2.3 PRE-DESIGN

The goal of the pre-design step is to convert a thorough understanding of the project into possible design alternatives. Brainstorming is an effective tool in this step. Here, a list of alternatives is developed. Since an embedded system typically involves both hardware and/or software, the designer can investigate whether requirements could be met with a hardware only solution or some combination of hardware and software. Generally, a hardware only solution executes faster;

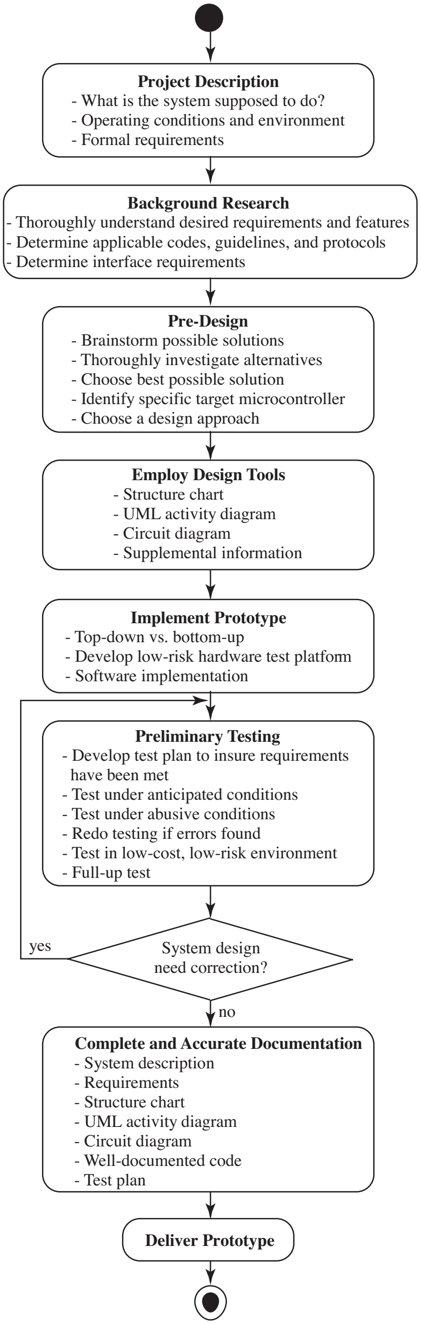

Project Description
- What is the system supposed to do?
- Operating conditions and environment
- Formal requirements
Background Research
- Thoroughly understand desired requirements and features
- Determine applicable codes, guidelines, and protocols
- Determine interface requirements
Pre-Design
- Brainstorm possible solutions
- Thoroughly investigate alternatives
- Choose best possible solution
- Identify specific target microcontroller
- Choose a design approach
Employ Design Tools
- Structure chart
- UML activity diagram
- Circuit diagram
- Supplemental information
Implement Prototype
- Top-down vs. bottom-up
- Develop low-risk hardware test platform
- Software implementation
Preliminary Testing
- Develop test plan to insure requirements have been met
- Test under anticipated conditions
- Test under abusive conditions
- Redo testing if errors found
- Test in low-cost, low-risk environment
- Full-up test
yes
System design need correction?
no
Complete and Accurate Documentation
- System description
- Requirements
- Structure chart
- UML activity diagram
- Circuit diagram
- Well-documented code
- Test plan
Deliver Prototype

Figure 8.1: Embedded system design process.

however, the design is fixed once fielded. On the other hand, a software implementation provides flexibility and a typically slower execution speed. Most embedded design solutions will use a combination of both hardware and software to capitalize on the inherent advantages of each.

Once a design alternative has been selected, the general partition between hardware and software can be determined. It is also an appropriate time to select a specific hardware device to implement the prototype design. If a microcontroller technology has been chosen, it is now time to select a specific controller. This is accomplished by answering the following questions.

- What microcontroller systems or features (i.e., ADC, PWM, timer, etc.) are required by the design?
- How many input and output pins are required by the design?
- What is the maximum anticipated operating speed of the microcontroller expected to be?

There are a wide variety of Microchip microcontrollers available to the designer.

8.2.4 DESIGN

With system requirements, controller features, partition between hardware and software, and a specific microcontroller determined, it is now time to tackle the actual design. It is important to follow a systematic and disciplined approach to design. This will allow for low stress development of a documented design solution that meets requirements. In the design step, several tools are employed to ease the design process. They include the following:

- employing a top-down design, bottom up implementation approach,
- using a structure chart to assist in partitioning the system,
- using a unified modeling language (UML) activity diagram to work out program flow, and
- developing a detailed circuit diagram of the entire system.

Let's take a closer look at each of these. The information provided here is an abbreviated version of the one provided in *Microcontrollers Fundamentals for Engineers and Scientists*. The interested reader is referred there for additional details and an in-depth example in Barrett and Pack [2008a].

Top-down design, bottom-up implementation. An effective tool to start partitioning the design is based on the techniques of top-down design, bottom-up implementation. In this approach, you start with the overall system and begin to partition it into subsystems. At this point of the design, you are not concerned with how the design will be accomplished but how the different pieces of the project will fit together. A handy tool to use at this design stage is the structure chart. The structure chart shows the hierarchy of how system hardware and software components will interact and interface with one another. You should continue partitioning system activity until each subsystem in the structure chart has a single definable function.

UML activity diagram. Once the system has been partitioned into pieces, the next step in the design process is to start working out the details of the operation of each subsystem we previously identified. Rather than beginning to code each subsystem as a function, we will work out the information and control flow of each subsystem using another design tool: the unified modeling language (UML) activity diagram. The activity diagram is simply a UML compliant flow chart. UML is a standardized method of documenting systems. The activity diagram is one of the many tools available from UML to document system design and operation. The basic symbols used in a UML activity diagram for a microcontroller based system are provided in Figure 8.2 [Fowler and Scott, 2000].

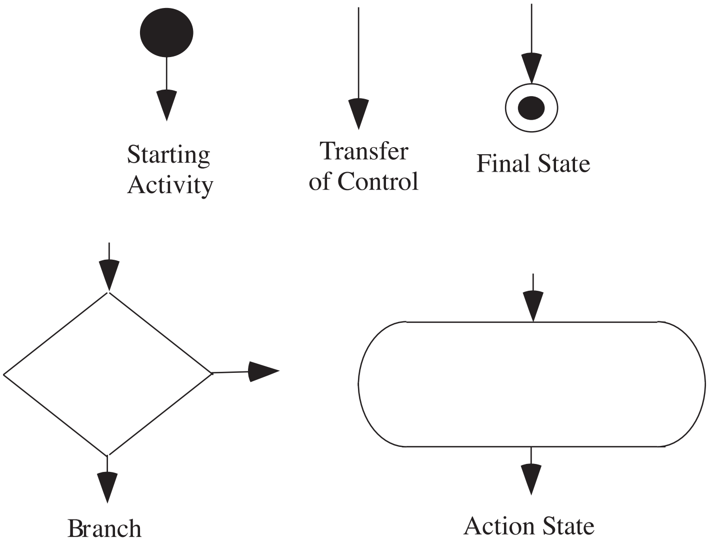


Figure 8.2: UML activity diagram symbols. (Adapted from Fowler and Scott [2000]).

To develop the UML activity diagram for the system, we can use a top-down, bottom-up, or a hybrid approach. In the top-down approach, we begin by modeling the overall flow of the algorithm from a high level. If we choose to use the bottom-up approach, we would begin at the bottom of the structure chart and choose a subsystem for flow modeling. The specific course of action chosen depends on project specifics. Often, a combination of both techniques, a hybrid approach, is used. You should work out all algorithm details at the UML activity diagram level prior to coding any software. If you cannot explain system operation at this higher level, first, you have no business being down in the detail of developing the code. Therefore, the UML activity diagram should be of sufficient detail so you can code the algorithm directly from it [Dale and Lilly, 1995].

In the design step, a detailed circuit diagram of the entire system is developed. It will serve as a roadmap to implement the system. It is also a good idea at this point to investigate available design information relative to the project. This would include hardware design examples, software code examples, and application notes available from manufacturers.

At the completion of this step, the prototype design is ready for implementation and testing.

8.2.5 IMPLEMENT PROTOTYPE

To successfully implement a prototype, an incremental approach should be followed. Again, the top-down design, bottom-up implementation provides a solid guide for system implementation. In an embedded system design involving both hardware and software, the hardware system including the microcontroller should be assembled first. This provides the software the required signals to interact with. As the hardware prototype is assembled on a prototype board, each component is tested for proper operation as it is brought online. This allows the designer to pinpoint malfunctions as they occur.

Once the hardware prototype is assembled, coding may commence. As before, software should be incrementally brought online. You may use a top-down, bottom-up, or hybrid approach depending on the nature of the software. The important point is to bring the software online incrementally such that issues can be identified and corrected early on.

It is highly recommended that low-cost stand-in components be used when testing the software with the hardware components. For example, push buttons, potentiometers, and LEDs may be used as low-cost stand-in component simulators for expensive input instrumentation devices and expensive output devices such as motors. This allows you to ensure the software is properly operating before using it to control the actual components.

8.2.6 PRELIMINARY TESTING

To test the system, a detailed test plan must be developed. Tests should be developed to verify that the system meets all of its requirements and also intended system performance in an operational environment. The test plan should also include scenarios in which the system is used in an unintended manner. As before a top-down, bottom-up, or hybrid approach can be used to test the system.

Once the test plan is completed, actual testing may commence. The results of each test should be carefully documented. As you go through the test plan, you will probably uncover a number of run time errors in your algorithm. After you correct a run time error, the entire test plan must be performed again. This ensures that the new fix does not have an unintended effect on another part of the system. Also, as you process through the test plan, you will probably think of other tests that were not included in the original test document. These tests should be added to the test plan. As you go through testing, realize your final system is only as good as the test plan that supports it!

Once testing is complete, you might try another level of testing where you intentionally try to "jam up" the system. In another words, try to get your system to fail by trying combinations of inputs that were not part of the original design. A robust system should continue to operate correctly in this type of an abusive environment. It is imperative that you design robustness into your system. When testing on a low-cost simulator is complete, the entire test plan should be performed again with the actual system hardware. Once this is completed you should have a system that meets its requirements!

8.2.7 COMPLETE AND ACCURATE DOCUMENTATION

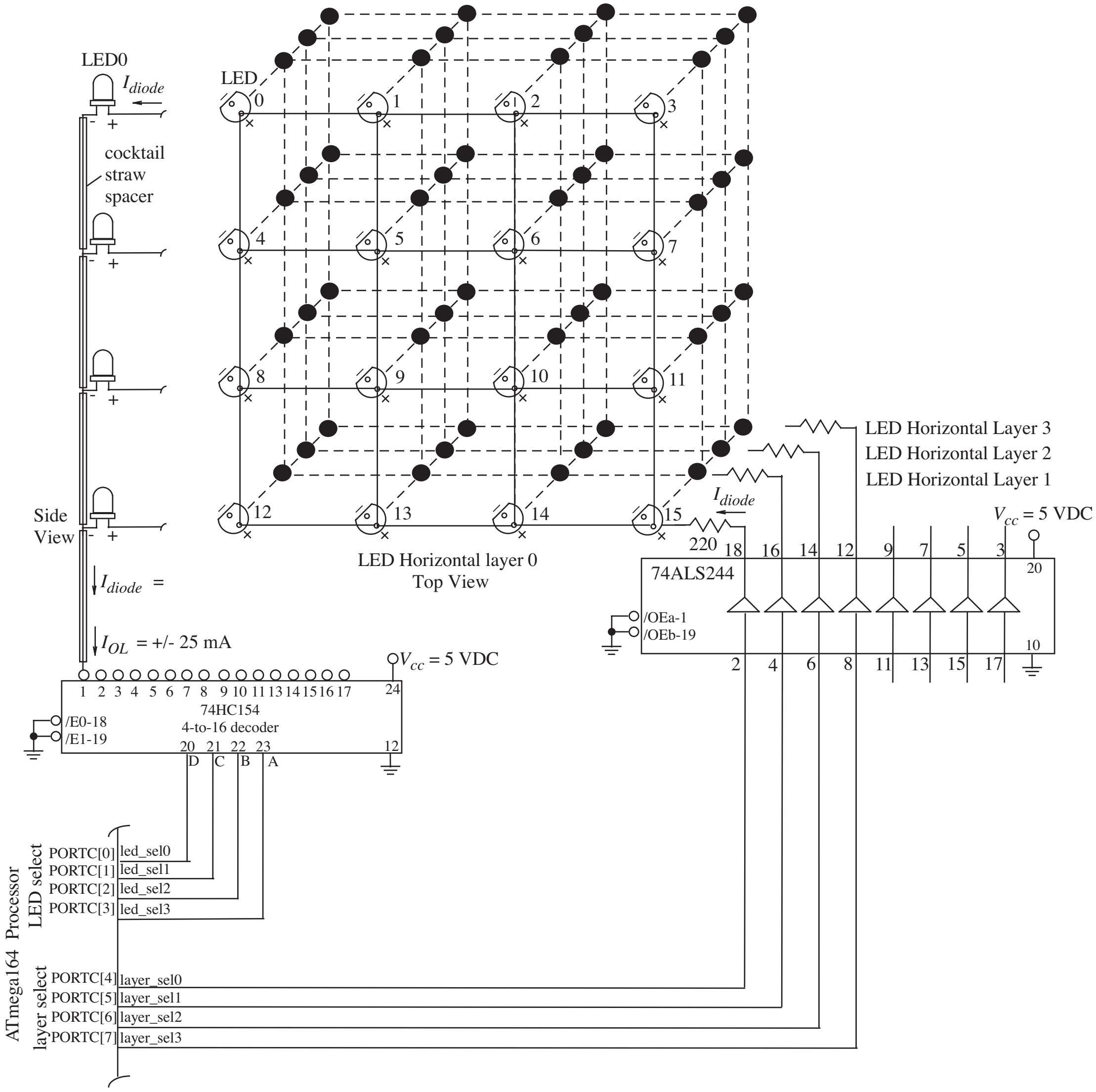
With testing complete, the system design should be thoroughly documented. Much of the documentation will have already been accomplished during system development. Documentation will include the system description, system requirements, the structure chart, the UML activity diagrams documenting program flow, the test plan, results of the test plan, system schematics, and properly documented code. To properly document code, you should carefully comment all functions describing their operation, inputs, and outputs. Also, comments should be included within the body of the function describing key portions of the code. Enough detail should be provided such that code operation is obvious. It is also extremely helpful to provide variables and functions within your code names that describe their intended use.

You might think that a comprehensive system documentation is not worth the time or effort to complete it. Complete documentation pays rich dividends when it is time to modify, repair, or update an existing system. Also, well-documented code may be often reused in other projects: a method for efficient and timely development of new systems.

8.3 SPECIAL EFFECTS LED CUBE

To illustrate some of the fundamentals of embedded system design and microcontroller interfacing, we construct a three-dimensional LED cube. This design was inspired by an LED cube kit available from Jameco (`www.jameco.com`). This design was originally provided in *Arduino Microcontroller—Processing for Everyone*. The design was adapted for use with the ATmega164 and used with permission of Morgan & Claypool.

The LED cube consists of four layers of LEDs with 16 LEDs per layer. Only a single LED is illuminated at a specific time. However, different effects may be achieved by how long a specific LED is left illuminated. A specific LED layer is asserted using the layer select pins on the microcontroller using a one-hot-code (a single line asserted while the others are de-asserted). The asserted line is fed through a 74HC244 (three state, octal buffer, line driver) which provides an I_{OH}/I_{OL} current of ± 35 mA, as shown in Figure 8.3. A given output from the 74HC244 is fed to a common anode connections for all 16 LEDs in a layer. All four LEDs in a specific LED position share a common cathode connection. That is, an LED in a specific location within a



Notes:

1. LED cube consists of 4 layers of 16 LEDs each.
2. Each LED is individually addressed by asserting the appropriate cathode signal (0–15) and asserting a specific LED layer.
3. All LEDs in a given layer share a common anode connection.
4. All LEDs in a given position (0–15) share a common cathode connection.

Figure 8.3: LED special effects cube.

layer shares a common cathode connection with three other LEDs that share the same position in the other three layers. The common cathode connection from each LED location is fed to a specific output of the 74HC154 4-to-16 decoder. The decoder has a one-cold-code output. To illuminate a specific LED, the appropriate layer select and LED select lines are asserted using the layer_sel[3:0] and led_sel[3:0] lines, respectively. This basic design may be easily expanded to a larger LED cube.

8.3.1 CONSTRUCTION HINTS

To limit project costs, low-cost red LEDs (Jameco #333973) were used. The project requires a total of 64 LEDs (4 layers of 16 LEDs each). A LED template pattern was constructed from a 5″ by 5″ piece of pine wood. A 4-by-4 pattern of holes were drilled into the wood. Holes were spaced 3/4″ apart. The hole diameter was slightly smaller than the diameter of the LEDs to allow for a snug LED fit.

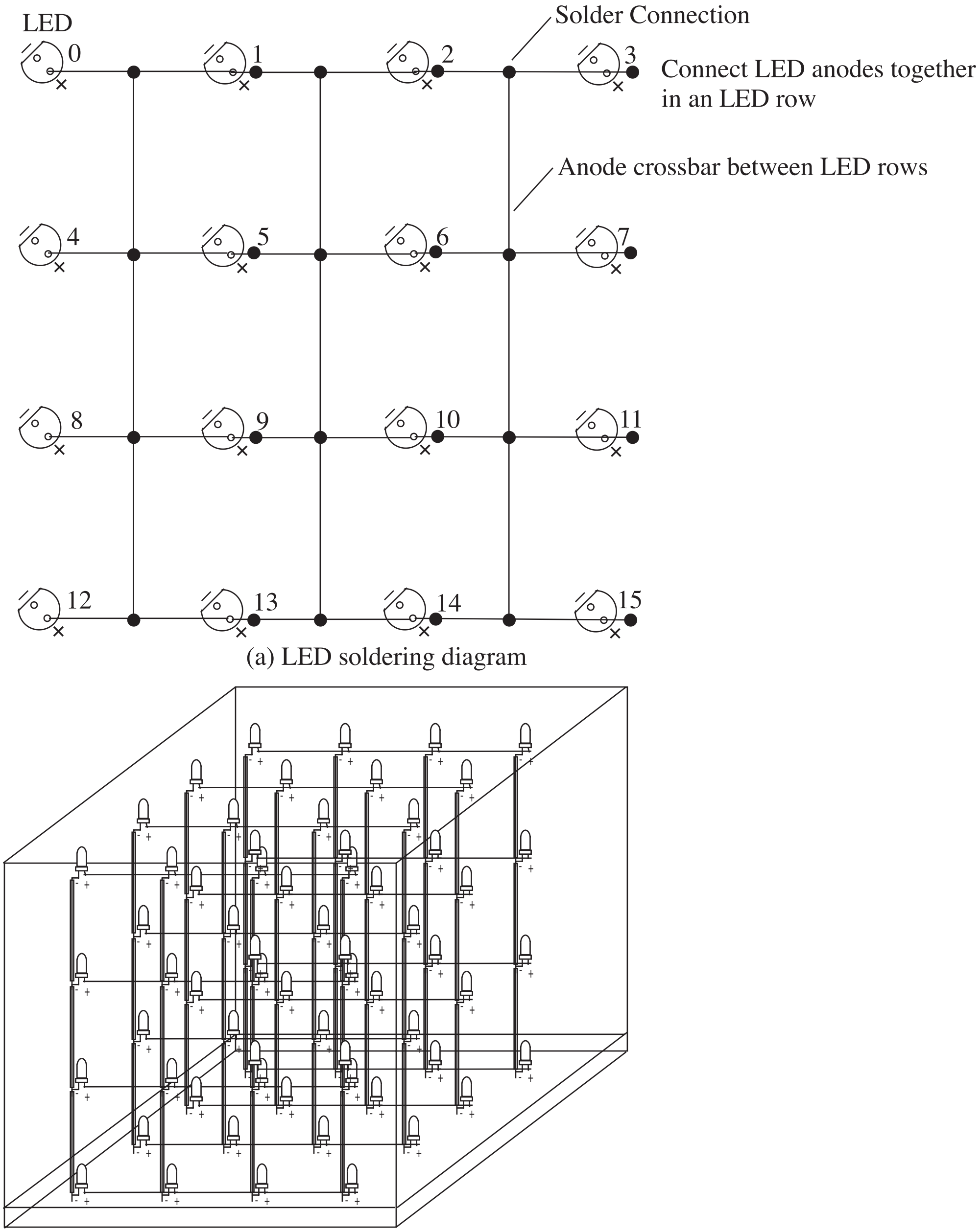
The LED array was constructed a layer at a time using the wood template. Each LED was tested before inclusion in the array. A 5 VDC power supply with a series 220 ohm resistor was used to ensure each LED was fully operational. The LED anodes in a given LED row were then soldered together. A fine tip soldering iron and a small bit of solder were used for each interconnect as shown in Figure 8.4. Cross wires were then used to connect the cathodes of adjacent rows. A 22 gage bare wire was used. Again, a small bit of solder was used for the interconnect points. Four separate LED layers (4 by 4 array of LEDs) were completed.

To assemble the individual layers into a cube, cocktail straw segments were used as spacers between the layers. The straw segments provided spacing between the layers and also offered improved structural stability. The anodes for a given LED position were soldered together. For example, all LEDs in position 0 for all four layers shared a common anode connection.

The completed LED cube was mounted on a perforated printed circuit board (perfboard) to provide a stable base. LED sockets for the 74LS244 and the 74HC154 were also mounted to the perfboard. Connections were routed to a 16-pin ribbon cable connector. The other end of the ribbon cable was interfaced to the appropriate pins of the ATmega164 processor. The entire LED cube was mounted within a 4″ plexiglass cube. The cube is available from the Container Store (`www.containerstore.com`). A construction diagram is provided in Figure 8.4.

8.3.2 LED CUBE CODE

In this example, a function "illuminate_LED" is used to illuminate a specific LED. The LED position (0–15), the LED layer (0–3), and the length of time to illuminate the LED in milliseconds are specified. In this short example, LED 0 is sequentially illuminated in each layer. An LED grid map is shown in Figure 8.5.


LED
0
1
2
3
Solder Connection
Connect LED anodes together in an LED row
Anode crossbar between LED rows
4
5
6
7
8
9
10
11
12
13
14
15

(a) LED soldering diagram

(b) 3D LED array mounted within plexiglass cube

Figure 8.4: LED cube construction.

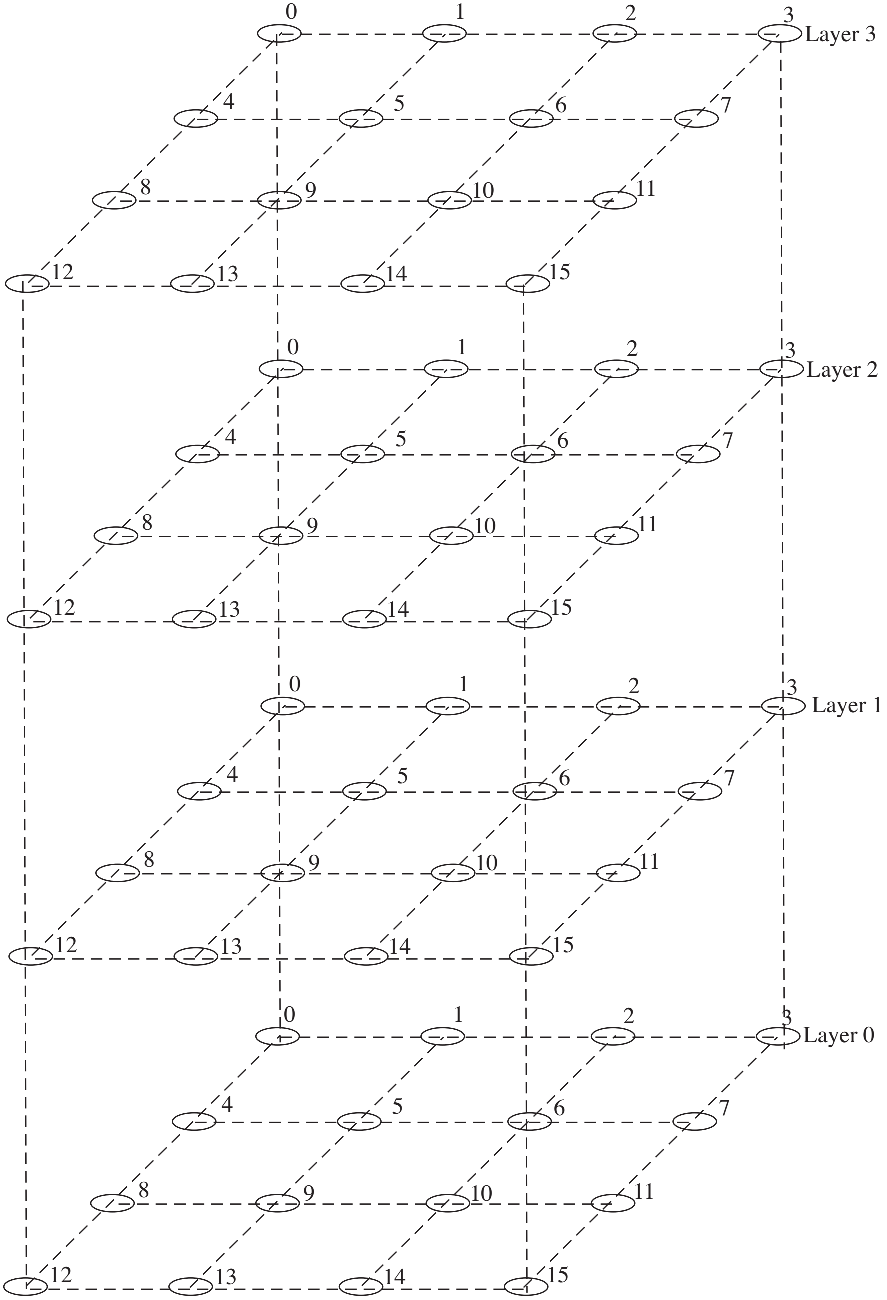


Figure 8.5: LED grid map.

```
//***************************************************************
//file name: ledcube1.c
//***************************************************************

//include files**************************************************

//MICROCHIP register definitions for ATmega164
#include <iom164pv.h>

//function prototypes********************************************
void delay(unsigned int number_of_26_2ms_interrupts);
void init_timer0_ovf_interrupt(void);
void initialize_ports(void);
void illuminate_LED(int led, int layer, int delay_time);

unsigned int   delay_timer = 0;

void main(void)
{
initialize_ports();                   //initialize ports
init_timer0_ovf_interrupt();          //init Timer0 for delay

while(1)
  {
                                      //specify delay in number
                                      //of 26.2 ms interrupts
  illuminate_LED(0, 0, 19);
  illuminate_LED(0, 1, 19);
  illuminate_LED(0, 2, 19);
  illuminate_LED(0, 3, 19);
  }
}

//***************************************************************
//function definitions*******************************************

//***************************************************************
//initialize_ports: provides initial configuration for I/O ports
//***************************************************************
```

```
void initialize_ports(void)
{
DDRA =0xff;  //set PORTA as output
PORTA=0x00;  //initialize low

DDRB =0x0f;  //PORTB[7:4] as input, set PORTB[3:0] as output
PORTB=0x00;  //disable PORTB pull-up resistors

DDRC =0xff; //set PORTC as output
PORTC=0x00; //initialize low

DDRD =0xff; //set PORTD as output
PORTD=0x00; //initialize low
}

//****************************************************************
//delay(unsigned int num_of_26_2_ms_interrupts): this generic
//delay function provides the specified delay as the number of
//26.2 ms "clock ticks" from the Timer0 interrupt.
//
//Note: this function is only valid when using a 2.5 MHz
//time base.  e.g., 20 MHz ceramic resonator divided by 8
//****************************************************************

void delay(unsigned int number_of_26_2ms_interrupts)
{
TCNT0 = 0x00;                        //reset delay_timer
delay_timer = 0;
while(delay_timer <= number_of_26_2ms_interrupts)
  {
  ;
  }
}

//****************************************************************
//int_timer0_ovf_interrupt(): The Timer0 overflow interrupt is
//being employed as a time base for a master timer for this
//project. The internal time base is set to operate at 2.5 MHz and
```

```
//then is divided by 256.  The 8-bit Timer0 register (TCNT0)
//overflows every 256 counts or every 26.2 ms.
//*****************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04; //divide timer0 timebase by 256, overflow occurs
               //every 26.2ms
TIMSK0 = 0x01; //enable timer0 overflow interrupt
asm("SEI");    //enable global interrupt
}

//*****************************************************************
//void timer0_interrupt_isr(void)
//*****************************************************************

void timer0_interrupt_isr(void)
{
delay_timer++;                            //increment timer
}

//*****************************************************************

void illuminate_LED(int led, int layer, int delay_time)
{
PORTC = 0x00;                            //turn off LEDs

if(led==0)
  {
  PORTC = PORTC | 0x00;
  }
else if(led==1)
  {
  PORTC = PORTC | 0x01;
  }
else if(led==2)
  {
  PORTC = PORTC | 0x02;
  }
```

```
else if(led==3)
  {
  PORTC = PORTC | 0x03;
  }
else if(led==4)
  {
  PORTC = PORTC | 0x04;
  }
else if(led==5)
  {
  PORTC = PORTC | 0x05;
  }
else if(led==6)
  {
  PORTC = PORTC | 0x06;
  }
else if(led==7)
  {
  PORTC = PORTC | 0x07;
  }
if(led==8)
  {
  PORTC = PORTC | 0x08;
  }
else if(led==9)
  {
  PORTC = PORTC | 0x09;
  }
else if(led==10)
  {
  PORTC = PORTC | 0x0a;
  }
else if(led==11)
  {
  PORTC = PORTC | 0x0b;
  }
else if(led==12)
  {
  PORTC = PORTC | 0x0c;
  }
else if(led==13)
  {
  PORTC = PORTC | 0x0d;
  }
else if(led==14)
  {
  PORTC = PORTC | 0x0e;
  }
else if(led==15)
  {
  PORTC = PORTC | 0x0f;
  }

if(layer==0)
  {
  PORTC = PORTC | 0x10;
  }
else if(layer==1)
  {
  PORTC = PORTC | 0x20;
  }
else if(layer==2)
  {
  PORTC = PORTC | 0x40;
  }
else if(layer==3)
  {
  PORTC = PORTC | 0x80;
  }

delay(delay_time);

PORTC = 0x00;                    //turn off LEDs
}

//**********************************************
```

In the next example, a "fireworks" special effect is produced. The firework goes up, splits into four pieces, and then falls back down as shown in Figure 8.6. It is useful for planning special effects.

```
//*************************************************************
//file name: ledcube2.c
//*************************************************************

//include files***********************************************

//MICROCHIP register definitions for ATmega164
#include <iom164pv.h>

//function prototypes*****************************************
void delay(unsigned int number_of_26_2ms_interrupts);
void init_timer0_ovf_interrupt(void);
void initialize_ports(void);
void illuminate_LED(int led, int layer, int delay_time);

unsigned int   delay_timer = 0;
unsigned int   i;

void main(void)
{
initialize_ports();                    //initialize ports
init_timer0_ovf_interrupt();           //init Timer0 for delay

while(1)
  {
                                       //specify delay in number
                                       //of 26.2 ms interrupts
  //firework going up
  illuminate_LED(5, 0, 4);
  illuminate_LED(5, 1, 4);
  illuminate_LED(5, 2, 4);
  illuminate_LED(5, 3, 4);

  //firework exploding into four pieces
  //at each cube corner
  for(i=0;i<=10;i++)
```

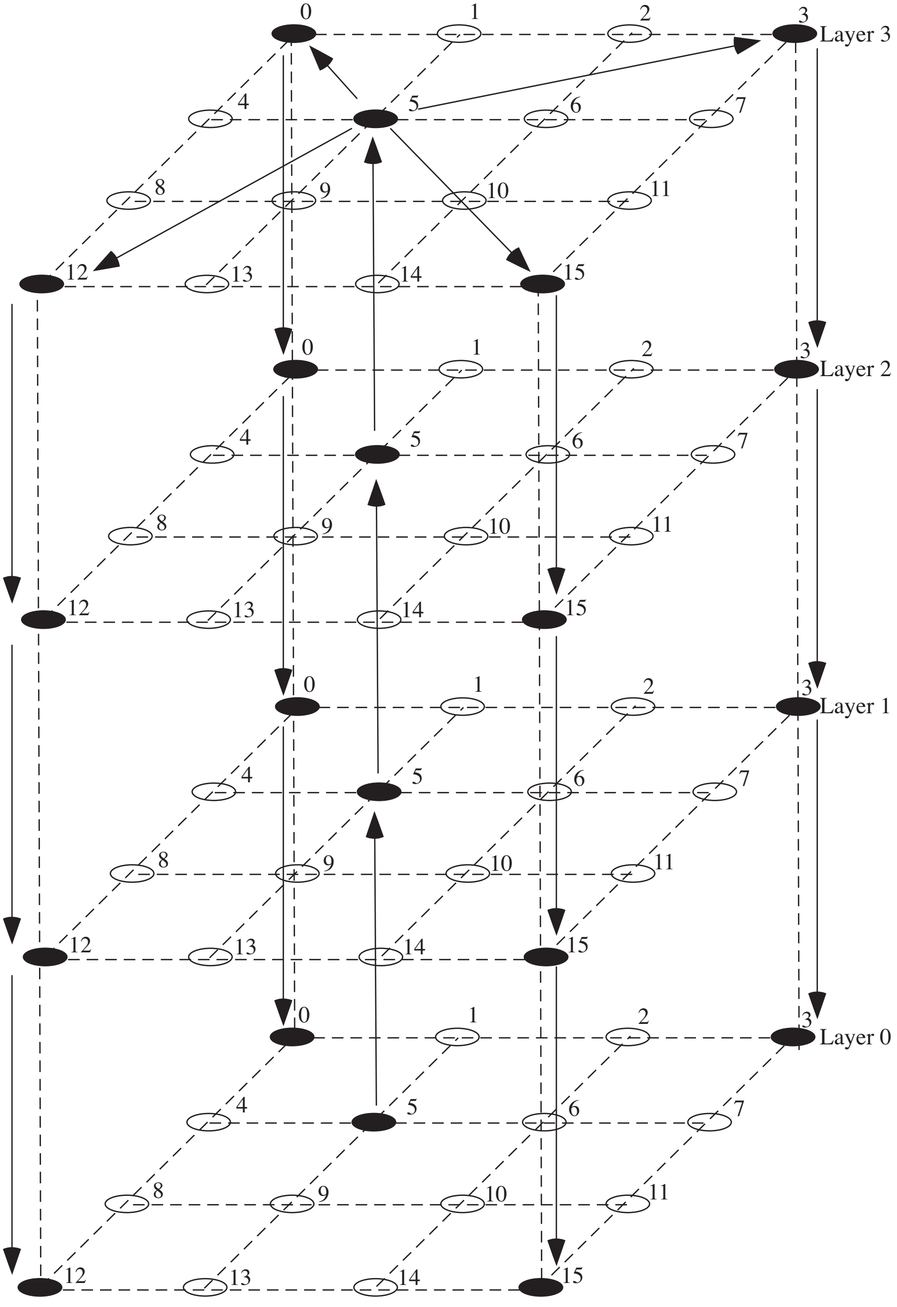


Figure 8.6: LED grid map for a fire work.

```
 {
 illuminate_LED(0,  3, 1);
 illuminate_LED(3,  3, 1);
 illuminate_LED(12, 3, 1);
 illuminate_LED(15, 3, 1);
 delay(1);
 }

delay(8);

//firework pieces falling to layer 2
for(i=0;i<=10;i++)
  {
  illuminate_LED(0,  2, 1);
  illuminate_LED(3,  2, 1);
  illuminate_LED(12, 2, 1);
  illuminate_LED(15, 2, 1);
  delay(1);
  }

delay(8);

//firework pieces falling to layer 1
for(i=0;i<=10;i++)
  {
  illuminate_LED(0,  1, 1);
  illuminate_LED(3,  1, 1);
  illuminate_LED(12, 1, 1);
  illuminate_LED(15, 1, 1);
  delay(1);
  }

delay(8);

//firework pieces falling to layer 0
for(i=0;i<=10;i++)
 {
 illuminate_LED(0,  0, 1);
 illuminate_LED(3,  0, 1);
```

```
    illuminate_LED(12, 0, 1);
    illuminate_LED(15, 0, 1);
    delay(1);
    }

   delay(1);

  }
}

//*************************************************************
//function definitions*****************************************

//*************************************************************
//initialize_ports: provides initial config for I/O ports
//*************************************************************

void initialize_ports(void)
{
DDRA =0xff;  //set PORTA as output
PORTA=0x00;  //initialize low

DDRB =0x0f;  //PORTB[7:4] as input, set PORTB[3:0] as output
PORTB=0x00;  //disable PORTB pull-up resistors

DDRC =0xff; //set PORTC as output
PORTC=0x00; //initialize low

DDRD =0xff; //set PORTD as output
PORTD=0x00; //initialize low
}

//*************************************************************
//delay(unsigned int num_of_26_2_ms_interrupts): this generic
//delay function provides the specified delay as the number of
//26.2 ms "clock ticks" from the Timer0 interrupt.
//
//Note: this function is only valid when using a 2.5 MHz
//time base.  e.g., 20 MHz ceramic resonator divided by 8
```

```
//**************************************************************

void delay(unsigned int number_of_26_2ms_interrupts)
{
TCNT0 = 0x00;                          //reset delay_timer
delay_timer = 0;
while(delay_timer <= number_of_26_2ms_interrupts)
  {
  ;
  }
}

//**************************************************************
//int_timer0_ovf_interrupt(): The Timer0 overflow interrupt is
//being employed as a time base for a master timer for this
//project. The internal time base is set to operate at 2.5 MHz
//and then is divided by 256.  The 8-bit Timer0 reg (TCNT0)
//overflows every 256 counts or every 26.2 ms.
//**************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04; //divide timer0 timebase by 256, overflow occurs
               //every 26.2ms
TIMSK0 = 0x01; //enable timer0 overflow interrupt
asm("SEI");    //enable global interrupt
}

//**************************************************************
//void timer0_interrupt_isr(void)
//**************************************************************

void timer0_interrupt_isr(void)
{
delay_timer++;                             //increment timer
}

//**************************************************************
```

```
void illuminate_LED(int led, int layer, int delay_time)
{
PORTC = 0x00;                               //turn off LEDs

if(led==0)
  {
  PORTC = PORTC | 0x00;
  }
else if(led==1)
  {
  PORTC = PORTC | 0x01;
  }
else if(led==2)
  {
  PORTC = PORTC | 0x02;
  }
else if(led==3)
  {
  PORTC = PORTC | 0x03;
  }
else if(led==4)
  {
  PORTC = PORTC | 0x04;
  }
else if(led==5)
  {
  PORTC = PORTC | 0x05;
  }
else if(led==6)
  {
  PORTC = PORTC | 0x06;
  }
else if(led==7)
  {
  PORTC = PORTC | 0x07;
  }
if(led==8)
  {
  PORTC = PORTC | 0x08;
  }
else if(led==9)
  {
  PORTC = PORTC | 0x09;
  }
else if(led==10)
  {
  PORTC = PORTC | 0x0a;
  }
else if(led==11)
  {
  PORTC = PORTC | 0x0b;
  }
else if(led==12)
  {
  PORTC = PORTC | 0x0c;
  }
else if(led==13)
  {
  PORTC = PORTC | 0x0d;
  }
else if(led==14)
  {
  PORTC = PORTC | 0x0e;
  }
else if(led==15)
  {
  PORTC = PORTC | 0x0f;
  }

if(layer==0)
  {
  PORTC = PORTC | 0x10;
  }
else if(layer==1)
  {
  PORTC = PORTC | 0x20;
  }
else if(layer==2)
```

```
  {
  PORTC = PORTC | 0x40;
  }
else if(layer==3)
  {
  PORTC = PORTC | 0x80;
  }

delay(delay_time);

PORTC = 0x00;                  //turn off LEDs
}

//*************************************************************
```

8.4 AUTONOMOUS MAZE NAVIGATING ROBOTS

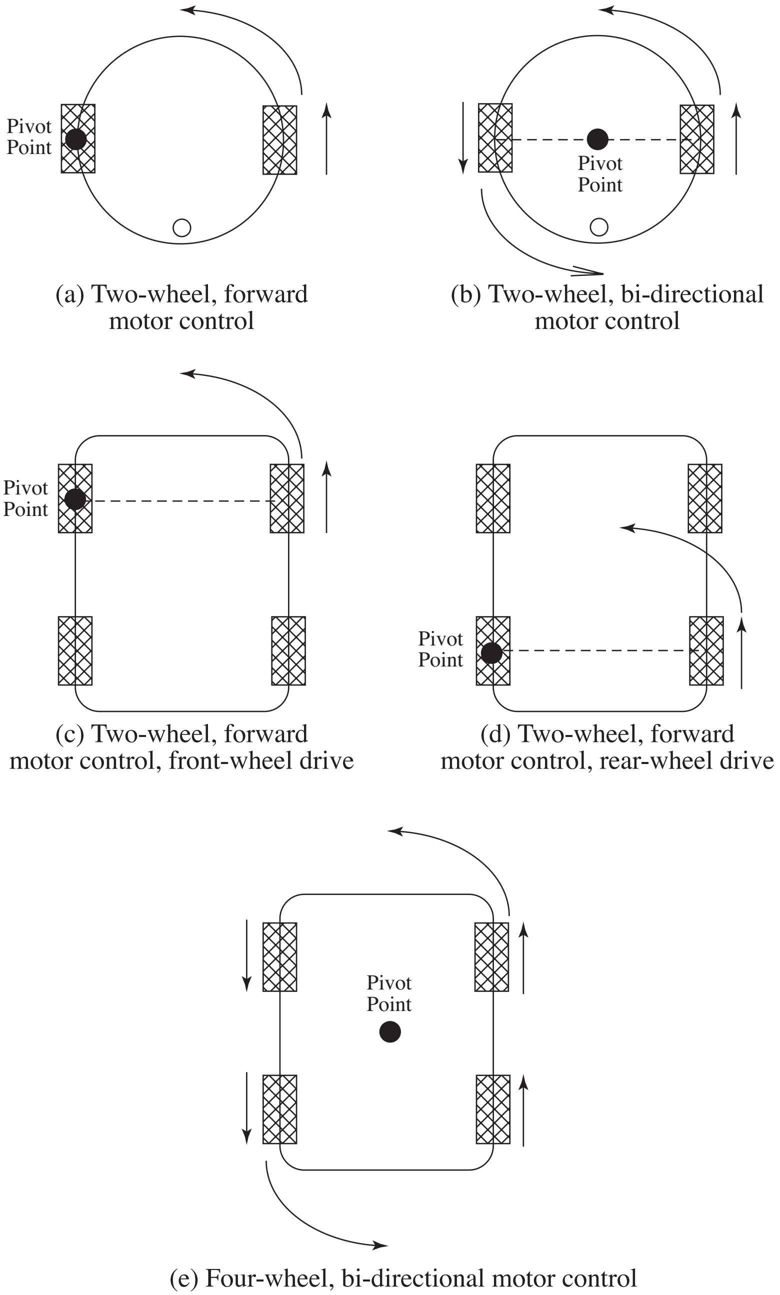
In the next several sections, we investigate two different autonomous navigating robot designs. Before delving into these designs, it would be helpful to review the fundamentals of robot steering and motor control. Figure 8.7 illustrates the fundamental concepts. Robot steering is dependent upon the number of powered wheels and whether the wheels are equipped with unidirectional or bidirectional control. Additional robot steering configurations are possible.

Recall from the previous chapter that an H-bridge is typically required for bidirectional control of a DC motor. An H-bridge configured for controlling a single motor is shown in Figure 8.8a. The H-bridge has two inputs: one for controlling motor direction and one for controlling motor speed via a pulse width modulated signal. Provided in Figure 8.8b is a motor control system for a two motor robot.

An autonomous, maze navigating robot is equipped with sensors to detect the presence of maze walls and navigate about the maze. The robot has no prior knowledge about the maze configuration. It uses the sensors and an onboard algorithm to determine the robot's next move. The overall goal is to navigate from the starting point of the maze to the end point as quickly as possible without bumping into maze walls as shown in Figure 8.9. Maze walls are usually painted white to provide a good, light reflective surface, whereas the maze floor is painted matte black to minimize light reflections.

8.4.1 DAGU MAGICIAN ROBOT

In this application project, we equip the Dagu Magician robot for control by ATmega164 as a maze navigating robot. The Magician kit may be purchased from SparkFun Electronics (`www.sparkfun.com`). The robot is controlled by two 7.2 VDC motors which independently drive a left and right wheel. A third non-powered drag ball provides tripod stability for the robot.



(a) Two-wheel, forward motor control

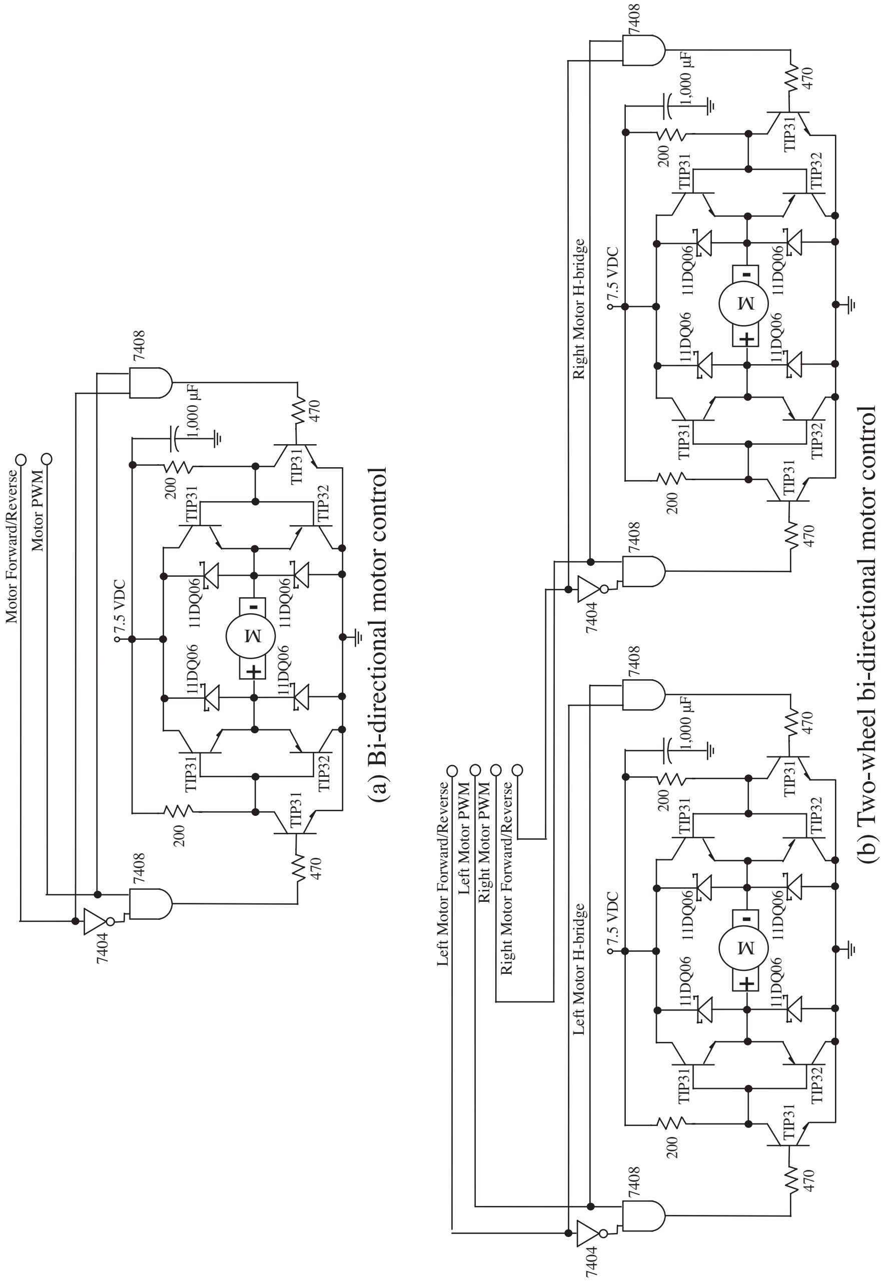
(b) Two-wheel, bi-directional motor control

(c) Two-wheel, forward motor control, front-wheel drive

(d) Two-wheel, forward motor control, rear-wheel drive

(e) Four-wheel, bi-directional motor control

Figure 8.7: Robot control configurations.



(a) Bi-directional motor control

(b) Two-wheel bi-directional motor control

Figure 8.8: Robot motor control.

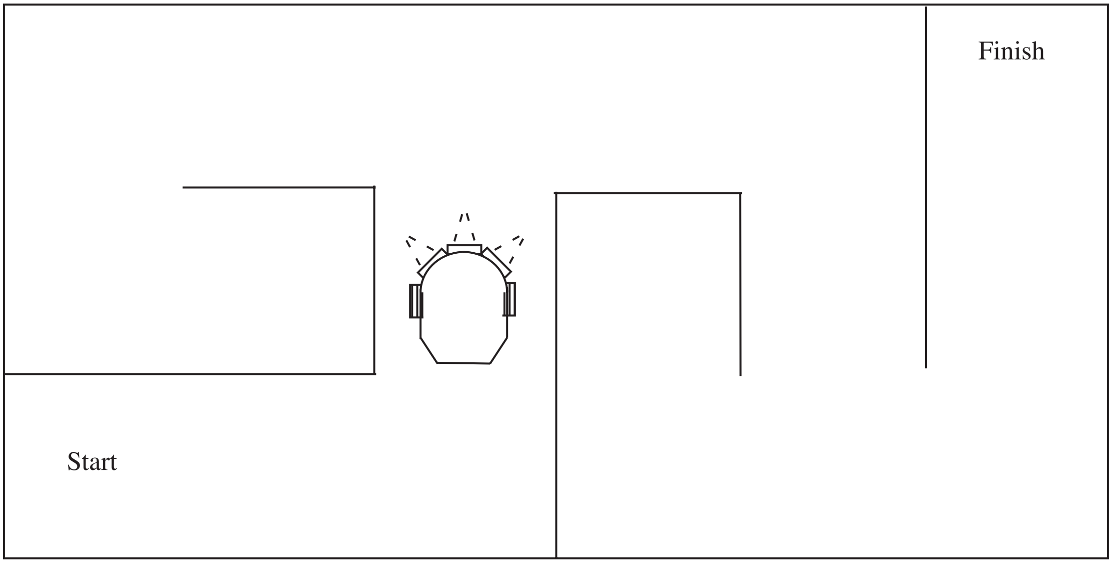


Figure 8.9: Autonomous robot within maze.

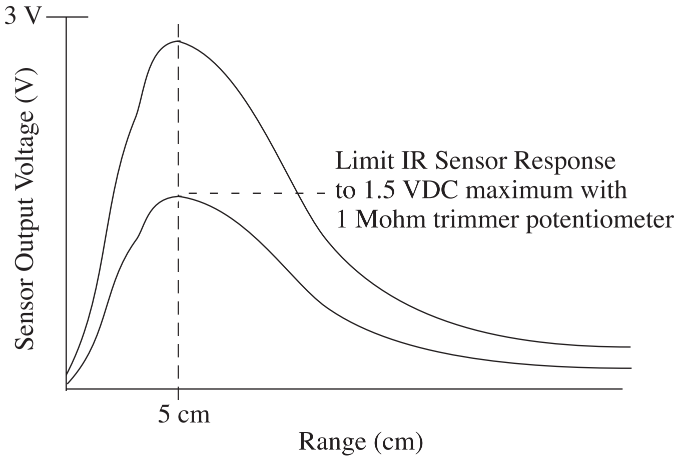


Figure 8.10: Sharp GP2Y0A21YKOF IR sensor profile.

We equip the Dagu Magician robot platform with three Sharp GP2Y0A21YKOF IR sensors as shown in Figure 8.11. The sensors are available from SparkFun Electronics (`www.sparkfun.com`). We mount the sensors on a bracket constructed from thin aluminum. Dimensions for the bracket are provided in the figure. Alternatively, the IR sensors may be mounted to the robot platform using "L" brackets available from a local hardware store. The characteristics of the

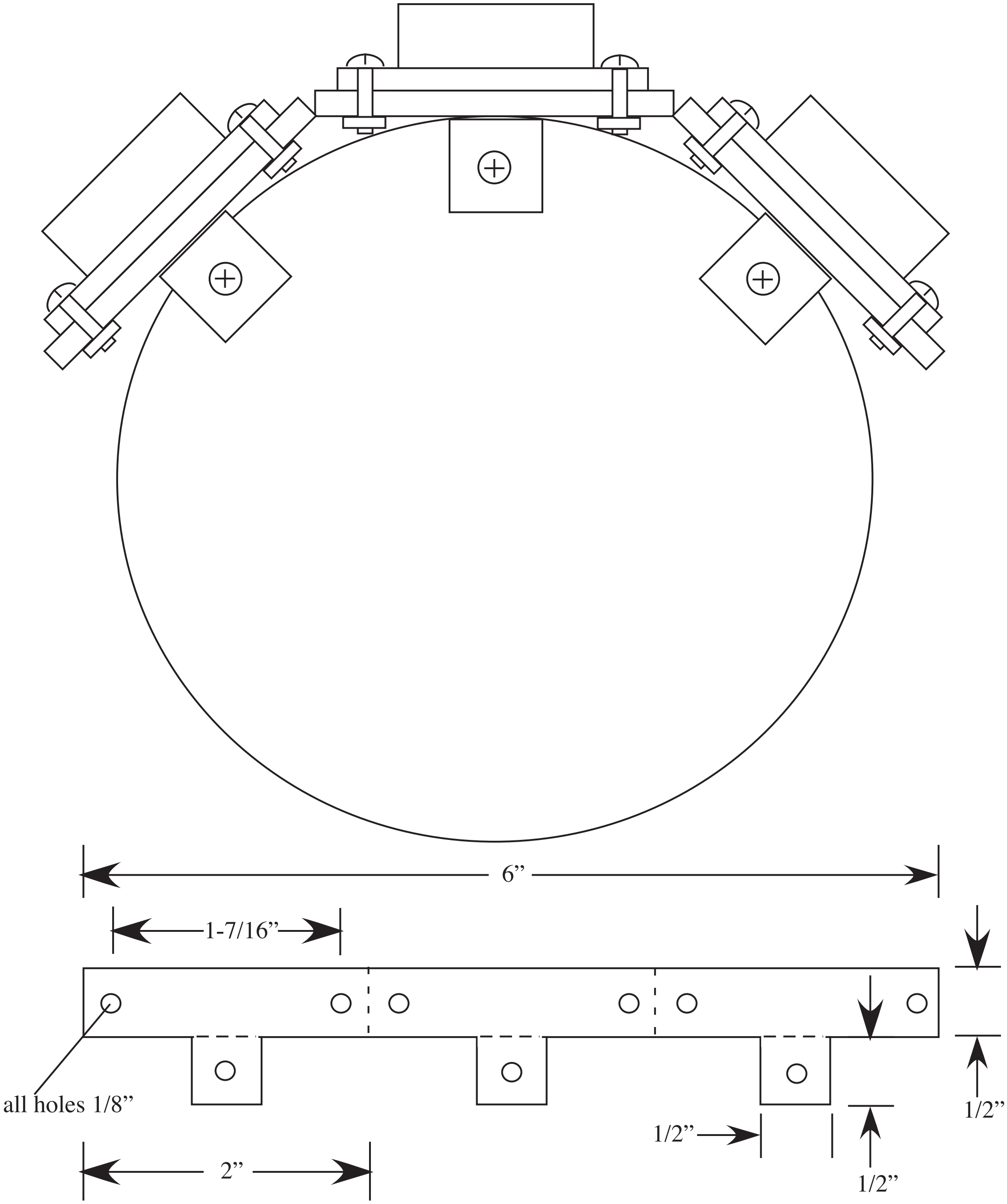


Figure 8.11: Dagu Magician robot platform modified with three IR sensors.

sensor are provided in Figure 8.10. The robot is placed in a maze with reflective walls. The project goal is for the robot to detect wall placement and navigate through the maze. It is important to note the robot is not provided any information about the maze. The control algorithm for the robot is hosted on ATmega164.

8.4.2 REQUIREMENTS

The requirements for this project are simple, the robot must autonomously navigate through the maze without touching maze walls. The robot motors may only be moved in the forward direction. We reviewed techniques to provide bi-directional motor control in Chapter 7. To render a left turn, the left motor is stopped and the right motor is asserted until the robot completes the turn. To render a right turn, the opposite action is required. The task in writing the control algorithm is to take the UML activity diagram provided in Figure 8.15 and the actions specified in the robot action truth table (Figure 8.12) and transform both into a coded algorithm. This may seem formidable but we take it a step at a time.

	Left Sensor	Middle Sensor	Right Sensor	Wall Left	Wall Middle	Wall Right	Left Motor	Right Motor	Left Signal	Right Signal	Comments
0	0	0	0	0	0	0	1	1	0	0	Forward
1	0	0	1	0	0	1	1	1	0	0	Forward
2	0	1	0	0	1	0	1	0	0	1	Right
3	0	1	1	0	1	1	0	1	1	0	Left
4	1	0	0	1	0	0	1	1	0	0	Forward
5	1	0	1	1	0	1	1	1	0	0	Forward
6	1	1	0	1	1	0	1	0	0	1	Right
7	1	1	1	1	1	1	1	0	0	1	Right

Figure 8.12: Truth table for robot action.

8.4.3 CIRCUIT DIAGRAM

The circuit diagram for the robot is provided in Figure 8.13. The three IR sensors (left, middle, and right) are mounted on the leading edge of the robot to detect maze walls. The output from the sensor is fed to three ADC channels (PORTA[2:0]). The robot motors will be driven by PWM channels A and B (OC1A and OC1B). The microcontroller is interfaced to the motors via a transistor with enough drive capability to handle the maximum current requirements of the motor. Since the microcontroller is powered at 5 VDC and the motors are rated at 3 VDC, two 1N4001 diodes are placed in series with the motor. This reduces the supply voltage to the motor to be approximately 3 VDC. The robot will be powered by a 9 VDC battery which is fed to a 5 VDC voltage regulator. Alternatively, a 9 VDC power supply rated at several amps may

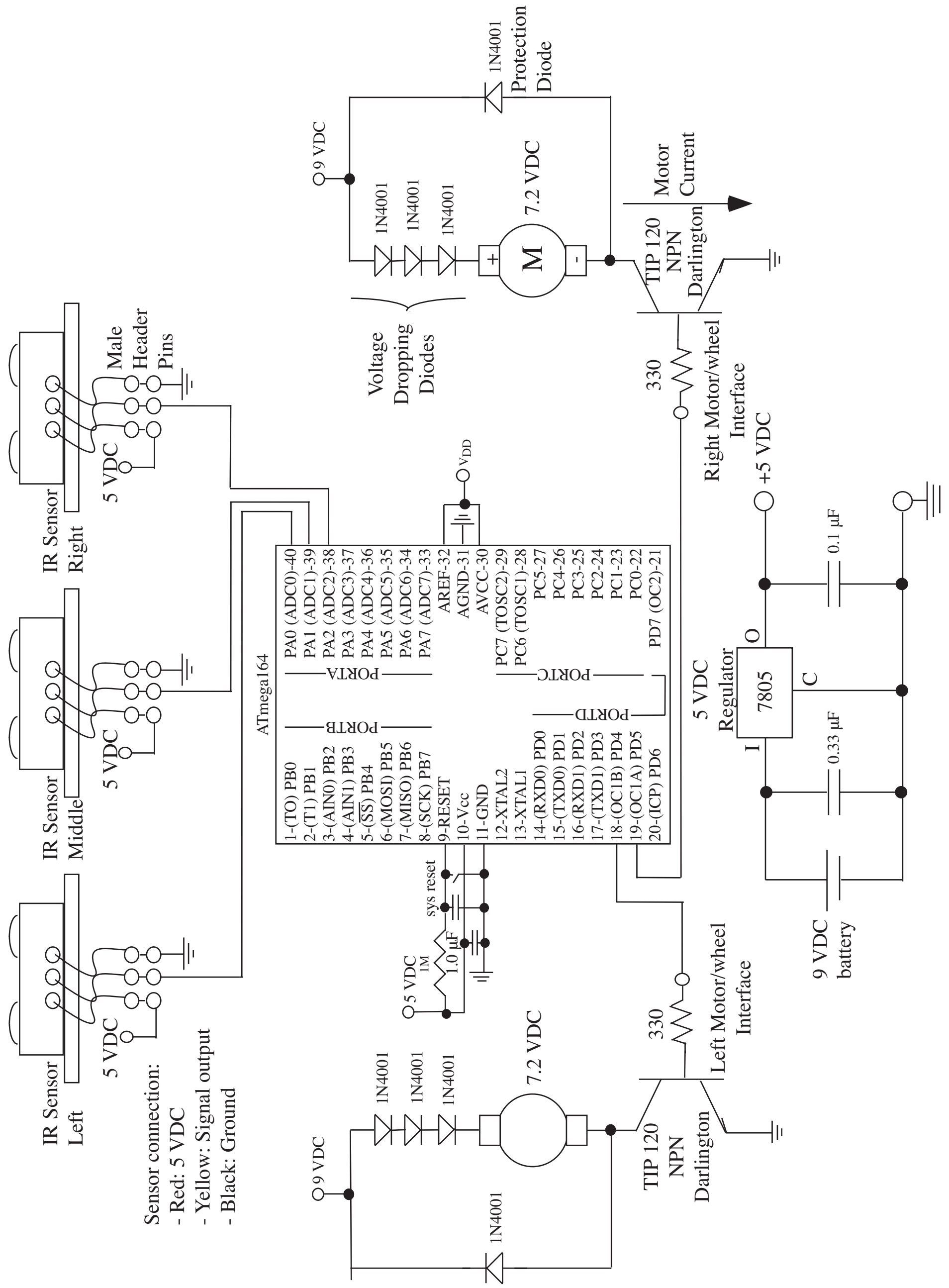


Figure 8.13: Robot circuit diagram.

be used in place of the 9 VDC battery. The supply may be connected to the robot via a flexible umbilical cable.

8.4.4 STRUCTURE CHART

The structure chart for the robot project is provided in Figure 8.14.

8.4.5 UML ACTIVITY DIAGRAMS

The UML activity diagram for the robot is provided in Figure 8.15.

8.4.6 MICROCONTROLLER CODE

Provided below is the basic framework for the code. As illustrated in the Robot UML activity diagram, the control algorithm initializes various ATmega164 subsystems (ports, ADC, and PWM), senses wall locations, and issues motor control signals to avoid walls.

It is helpful to characterize the infrared sensor response to the maze walls. This allows a threshold to be determined indicating the presence of a wall. In this example, we assume that a threshold of 2.5 VDC has been experimentally determined.

It is important to note that the amount of robot turn is determined by the PWM duty cycle (motor speed) and the length of time the turn is executed. For motors without optical

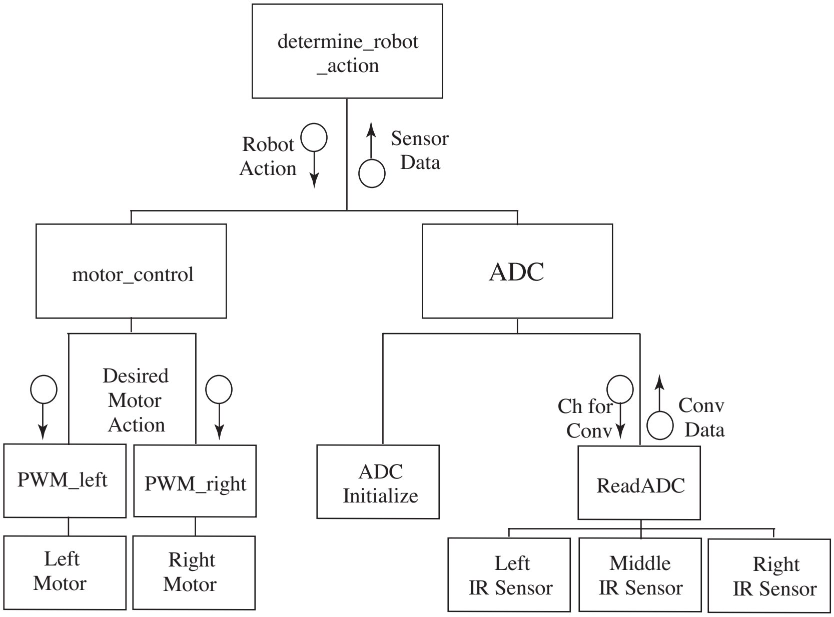


Figure 8.14: Robot structure diagram.

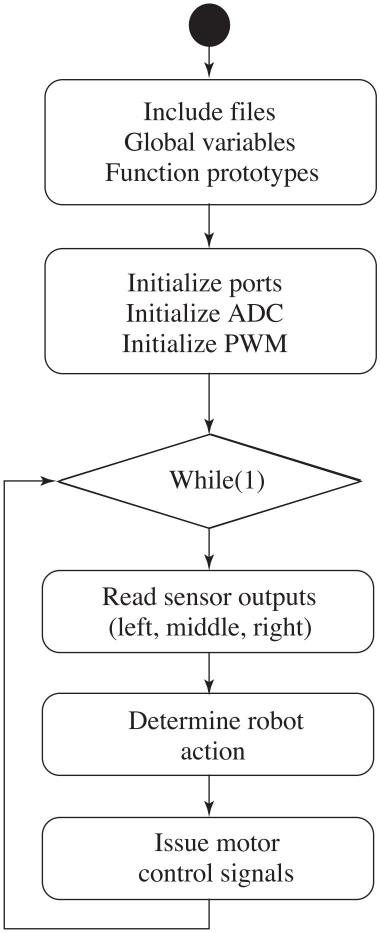


Figure 8.15: Robot UML activity diagram.

tachometers, the appropriate values for duty cycle and motor on time must be experimentally determined. In the example functions provided, the motor PWM and on time are fixed. Functions, where the motor duty cycle and on time are passed to the function as local variables, are left as a homework assignment.

```
//*************************************************************
#include<iom164v.h>                  //ATmega164 include files
                                     //function prototypes
void init_ADC(void);
unsigned int Read_ADC(unsigned char channel);
void PWM(unsigned char Duty_Cycle_Left, unsigned char Duty_Cycle_Right);
void ADC_values(void);
void PWM_forward(void);
void PWM_left(void);
void PWM_right(void);
void delay(unsigned int number_of_8_192ms_interrupts);
void init_timer2_ovf_interrupt(void);
void timer2_interrupt_isr(void);
void initialize_ports(void);

//interrupt handler definition
#pragma interrupt_handler timer2_interrupt_isr:12

//Global variables
float left_IR_voltage   = 0.0;
float right_IR_voltage  = 0.0;
float center_IR_voltage = 0.0;
unsigned int input_delay;

void main(void)
{
initialize_ports();                //initialize ports
init_timer2_ovf_interrupt();       //initialize interrupts
init_ADC();                        //initialize ADC

while(1)
  {
  ADC_values();
  determine_robot_action();
  }
}

//**************************************************************
// void initialize_ports(void) //   1: output, 0: input
//*************************************************************

void initialize_ports(void)
{
DDRA = 0xF8;                       //PORTA[2:0] input
DDRB = 0xFF;
DDRC = 0xFF;
DDRD = 0xFF;
}

//*************************************************************
//void determine_robot_action(void)
//In this example we assume that a threshold of 2.5 VDC has been
//experimentally determined.
//*************************************************************

void determine_robot_action(void)
{
                                 //wall on left and front, turn right
if((left_IR_voltage >= 2.5)&&(center_IR_voltage >= 2.5 VDC)
  &&(right_IR_voltage < 2.5))
  {
  PWM_right();
  }
else if                          //provide other cases here

  :
  :
  :
}

//***************************************************************
//void ADC_values(void)
//
//  PORTA[0] - Left IR Sensor
//  PORTA[1] - Center IR Sensor
//  PORTA[2] - Right IR Sensor
//***************************************************************

void ATD_values(void)
{
left_IR_Voltage   = (Read_ADC(0)*5.0)/1024.0;
center_IR_Voltage = (Read_ADC(1)*5.0)/1024.0;
right_IR_Voltage  = (Read_ADC(2)*5.0)/1024.0;
}
```

```
//***************************************************************
//void PWM_forward(void): the PWM is configured to make the
//motors go forward.
//Implementation notes:
//  - The left motor is controlled by PWM channel OC1B
//  - The right motor is controlled by PWM channel OC1A
//  - To go forward the same PWM duty cycle is applied to both
//    the left and right motors.
//  - The length of the delay controls the amount of time the
//    motors are powered.
//***************************************************************

void PWM_forward(void) {

TCCR1A = 0xA1;               //freq = resonator/510 = 10 MHz/510
                             //freq = 19.607 kHz
TCCR1B = 0x01;               //no clock source division
                             //Initiate PWM duty cycle variables

                             //Set PWM for left and right motors
                             //to 50
OCR1BH = 0x00;
                             //PWM duty cycle CH B left motor
OCR1BL = (unsigned char)(128);
OCR1AH = 0x00;                //PWM duty cycle CH B right motor
OCR1AL = (unsigned char)(128);
delay(122);                   //delay 1s
OCR1BL = (unsigned char)(0);  //motors off
OCR1AL = (unsigned char)(0);
}

//*************************************************************
//void PWM_left(void)
//Implementation notes:
//  - The left motor is controlled by PWM channel OC1B
//  - The right motor is controlled by PWM channel OC1A
//  - To go left the left motor is stopped and the right motor
//    is provided a PWM signal.  The robot will pivot about
//    the left motor.
//  - The length of the delay controls the amount of time the
//    motors are powered.
//***************************************************************

void PWM_left(void)
{
TCCR1A = 0xA1;                 //freq = resonator/510 = 10 MHz/510
                               //freq = 19.607 kHz
TCCR1B = 0x01;                 //no clock source division
                               //Initiate PWM duty cycle variables
                               //Set PWM for left motor at 0
                               //and the right motor to 50
OCR1BH = 0x00;                 //PWM duty cycle CH B left motor
OCR1BL = (unsigned char)(0);
OCR1AH = 0x00;                 //PWM duty cycle CH B right motor
OCR1AL = (unsigned char)(128);
delay(122);                    //delay 1 sec
OCR1BL = (unsigned char)(0);   //motors off
OCR1AL = (unsigned char)(0);
}

//****************************************************************
// void PWM_right(void)
//  - The left motor is controlled by PWM channel OC1B
//  - The right motor is controlled by PWM channel OC1A
//  - To go right the right motor is stopped and the left motor is
//    provided a PWM signal. The robot will pivot about the right
//    motor.
//  - The length of the delay controls the amount of time the
//    motors are powered.
//*****************************************************************

void PWM_right(void)
{
TCCR1A = 0xA1;                   //freq = resonator/510 = 10 MHz/510
                                 //freq = 19.607 kHz
TCCR1B = 0x01;                   //no clock source division
                                 //Initiate PWM duty cycle variables
                                 //Set PWM for left motor to 50
                                 //and right motor to 0
OCR1BH = 0x00;
                                 //PWM duty cycle CH B left motor
OCR1BL = (unsigned char)(128);
OCR1AH = 0x00;                   //PWM duty cycle CH B right motor
OCR1AL = (unsigned char)(0);
delay(122);                      //delay 1 sec
OCR1BL = (unsigned char)(0);     //motors off
OCR1AL = (unsigned char)(0);
}

//***************************************************************
void Init_ADC(void)
{
ADMUX = 0;                     //Select channel 0
ADCSRA = 0xC3;                 //Enable ADC & start 1st
                               //dummy conversion
                               //Set ADC module prescalar to 8
                               //critical for accurate ADC results
while (!(ADCSRA & 0x10));      //Check if conversation is ready
ADCSRA |= 0x10;                //Clear conv rdy flag - set the bit
}

//****************************************************************
unsigned int Read_ADC(unsigned char channel)
{
unsigned int binary_weighted_voltage = 0x00;
unsigned intbinary_weighted_voltage_low = 0x00;
unsigned int binary_weighted_voltage_high = 0x00;

ADMUX = channel;               //Select channel
ADCSRA |= 0x43;                //Start conversion
                               //Set ADC module prescalar to 8
                               //critical for accurate ADC results
while (!(ADCSRA & 0x10));      //Check if conversion is ready
ADCSRA |= 0x10;                //Clear Conv rdy flag - set the bit
```

```
binary_weighted_voltage_low = ADCL; //Read 8 low bits first
                               //Read 2 high bits, multiply by 256
                               //Shift to the left 8 times to get
                               //the upper "ADC" result
                               //into the correct position to be
                               //ORed with the Lower result.

binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
                               //Cast to unsigned int
                               //OR the two results together
                               //to form the 10 bit result
binary_weighted_voltage = binary_weighted_voltage_low
                        | binary_weighted_voltage_high;

return binary_weighted_voltage; //ADCH:ADCL
}

//*************************************************************
void delay(unsigned int number_of_8_192ms_interrupts)
{
TCNT2 = 0x00;                  //reset timer2
input_delay = 0;               //reset timer2 overflow counter
while(input_delay <= number_of_8_192ms_interrupts)
  {
  ;                            //wait specified number of interrupts
  } }

//**************************************************************
void init_timer2_ovf_interrupt(void)
{
TCCR2A = 0x00;   //Do nothing with this reg, not needed for count
TCCR2B = 0x06;   //div timer2 timebase by 256, overflow at 8.192ms
TIMSK2 = 0x01;   //enable timer2 overflow interrupt
asm("SEI");      //enable global interrupt
}

//***************************************************************

void timer2_interrupt_isr(void)
```

```
{
input_delay++;                    //increment overflow counter
}

//*************************************************************
```

Testing the control algorithm: It is recommended that the algorithm be first tested without the entire robot platform. This may be accomplished by connecting the three IR sensors and LEDS to the appropriate pins on the ATmega164 as specified in Figure 8.13. In place of the two motors and their interface circuits, two LEDs with the required interface circuitry may be used. The LEDs will illuminate to indicate the motors would be on during different test scenarios. Once this algorithm is fully tested in this fashion, the ATmega164 may be mounted to the robot platform and connected to the motors. Full-up testing in the maze may commence. Enjoy!

The design provided is very basic. As an end of chapter homework assignments, extend the design to include the following.

- Modify the PWM turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as variables to the function.
- Equip the motor with another IR sensor that looks down to the maze floor for "land mines." A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate it by rotating three times and flashing a large LED while rotating.
- Develop a function for reversing the robot.

8.5 MOUNTAIN MAZE NAVIGATING ROBOT

In this project we extend the maze navigating project to a three-dimensional mountain pass. Also, we use a robot equipped with four motorized wheels. Each of the wheels is equipped with an H-bridge to allow bidirectional motor control. In this example we will only control two wheels. We leave the development of a 4WD robot as an end of chapter homework assignment.

8.5.1 DESCRIPTION

For this project a DF Robot 4WD mobile platform kit was used (DFROBOT ROB0003, Jameco #2124285). The robot kit is equipped with four powered wheels. As in the Dagu Magician project, we equipped the DF Robot with three Sharp GP12D IR sensors, as shown in Figure 8.16. The robot will be placed in a three-dimensional maze with reflective walls modeled after a mountain pass. The goal of the project is for the robot to detect wall placement and navigate through the maze. The robot will not be provided any information about the maze. The control algorithm for the robot is hosted on the ATmega164.

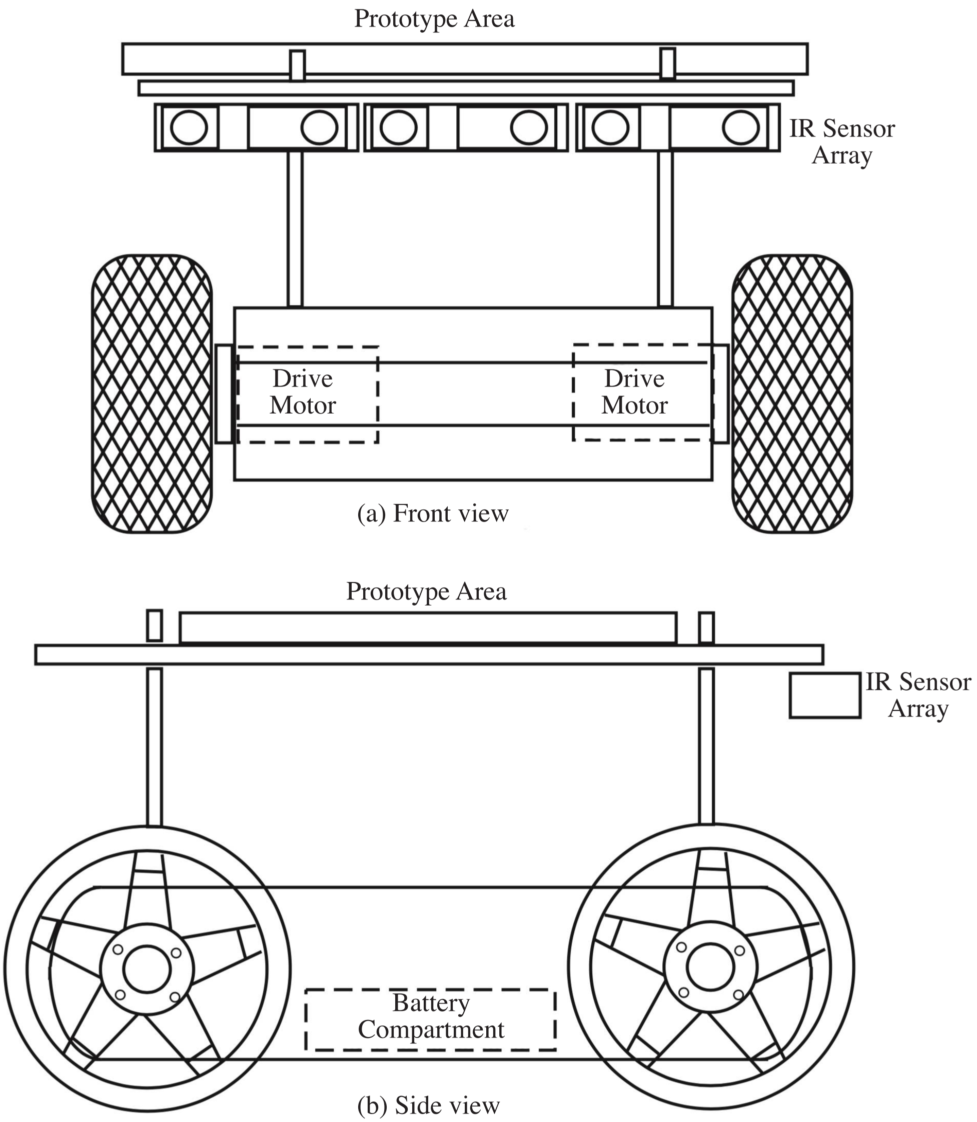


Figure 8.16: Robot layout.

8.5.2 REQUIREMENTS

The requirements for this project are simple, the robot must autonomously navigate through the maze without touching maze walls.

8.5.3 CIRCUIT DIAGRAM

The circuit diagram for the robot is provided in Figure 8.17. The three IR sensors (left, middle, and right) will be mounted on the leading edge of the robot to detect maze walls. The output from the sensor is fed to three ADC channels (PORTA[2:0]). The robot motors will be driven by PWM channels A and B (OC1A and OC1B) via an H-bridge. The robot is powered by a 7.5 VDC battery pack (5 AA batteries) which is fed to a 5 VDC voltage regulator. Alternatively, the robot may be powered by a 7.5 VDC power supply rated at several amps. In this case, the power is delivered to the robot by a flexible umbilical cable.

8.5.4 STRUCTURE CHART

The structure chart for the robot project is provided in Figure 8.18.

8.5.5 UML ACTIVITY DIAGRAMS

The UML activity diagram for the robot is provided in Figure 8.19.

8.5.6 MICROCONTROLLER CODE

The code for the robot may be adapted from that for the Dagu Magician robot. Since the motors are equipped with an H-bridge, slight modifications are required to the robot turning code. These modifications are provided below.

```
//**************************************************************
//void PWM_forward(void): the PWM is configured to make the
//motors go forward.
//Implementation notes:
//  - The left motor is controlled by PWM channel OC1B
//  - The right motor is controlled by PWM channel OC1A
//  - To go forward the same PWM duty cycle is applied to both
//    the left and right motors.
//  - The length of the delay controls the amount of time the
//    motors are powered.
//  - Direction control for the right motor is provided on
//    PORTD[6] 1: forward, 0: reverse
//  - Direction control for the left motor is provided on
//    PORTD[3] 1: forward, 0: reverse
/****************************************************************
```

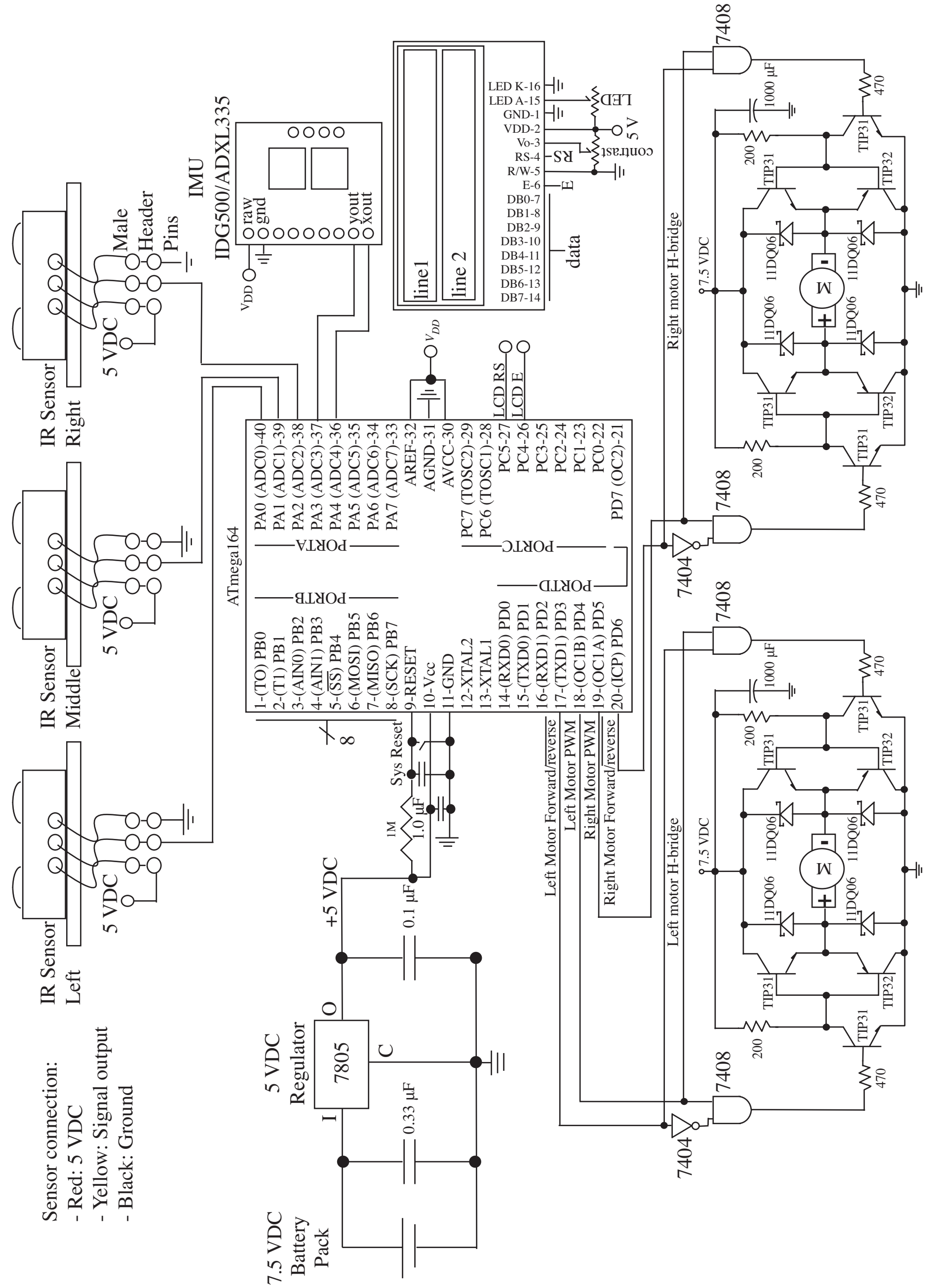


Figure 8.17: Robot circuit diagram.

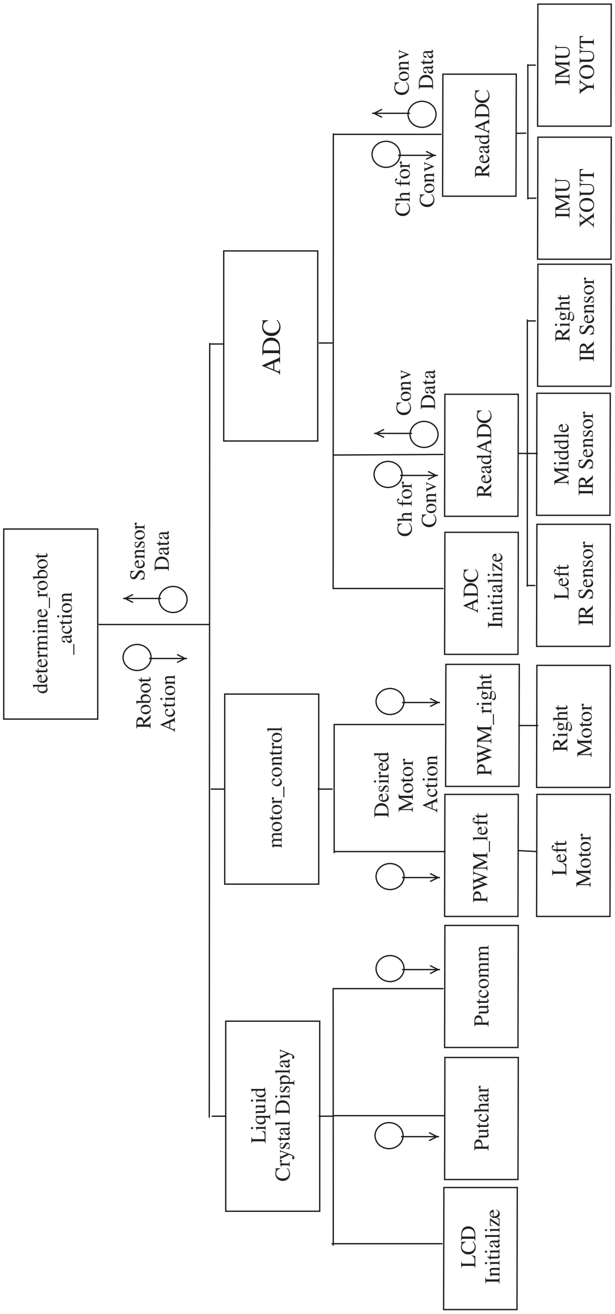


Figure 8.18: Robot structure diagram.

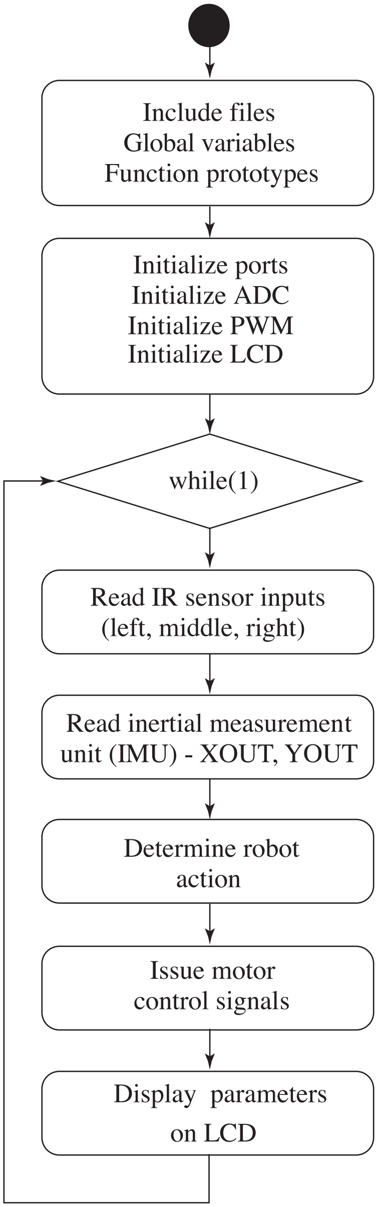


Figure 8.19: Robot UML activity diagram.

```
void PWM_forward(void)
{

TCCR1A = 0xA1;                //freq = resonator/510 = 10 MHz/510
                              //freq = 19.607 kHz
TCCR1B = 0x01;                //no clock source division

PORTD = PORTD | 0x48;         //Right and left motor forward
                              //PORTD[6]=1 and PORTD[3]=1
                              // 0 1 0 0 _ 1 0 0 0
                              //             0x48
                              //Initiate PWM duty cycle variables
                              //Set PWM for left and right motors
                              //to 50
OCR1BH = 0x00;                //PWM duty cycle CH B left motor
OCR1BL = (unsigned char)(128);
OCR1AH = 0x00;                //PWM duty cycle CH B right motor
OCR1AL = (unsigned char)(128);
delay(122);                   //delay 1 sec
OCR1BL = (unsigned char)(0); //motors off - no PWM duty cycle
OCR1AL = (unsigned char)(0);
}

//*****************************************************************
//void PWM_left(void)
//Implementation notes:
//  - The left motor is controlled by PWM channel OC1B
//  - The right motor is controlled by PWM channel OC1A
//  - To go left the left motor is stopped and the right motor is
//    provided a PWM signal.  The robot will pivot about the left
//    motor.
//  - The length of the delay controls the amount of time the
//    motors are powered.
//  - Direction control for the right motor is provided on PORTD[6]
//    1: forward, 0: reverse
//  - Direction control for the left motor is provided on PORTD[3]
//    1: forward, 0: reverse
//******************************************************************
```

```
void PWM_left(void)
{
TCCR1A = 0xA1;               //freq = resonator/510 = 10 MHz/510
                             //freq = 19.607 kHz
TCCR1B = 0x01;               //no clock source division

PORTD = PORTD & 0xF7;        //Left motor reverse  - PORTD[3]=0
PORTD = PORTD | 0x40;        //Right motor forward - PORTD[6]=1

                             //Initiate PWM duty cycle variables
                             //Set PWM for left motor at 0
                             //and the right motor to 50
OCR1BH = 0x00;               //PWM duty cycle CH B left motor
OCR1BL = (unsigned char)(128);
OCR1AH = 0x00;               //PWM duty cycle CH B right motor
OCR1AL = (unsigned char)(128);
delay(122);                  //delay 1s
OCR1BL = (unsigned char)(0); //motors off
OCR1AL = (unsigned char)(0);
}

//**************************************************************
// void PWM_right(void)
//  - The left motor is controlled by PWM channel OC1B
//  - The right motor is controlled by PWM channel OC1A
//  - To go right the right motor is stopped and the left motor is
//    provided a PWM signal.
The robot will pivot about the right motor.
//  - The length of the delay controls the amount of time the motors
//    are powered.
//  - Direction control for the right motor is provided on PORTD[6]
//    1: forward, 0: reverse
//  - Direction control for the left motor is provided on PORTD[3]
//    1: forward, 0: reverse
//***************************************************************

void PWM_right(void)
{
```

```
TCCR1A = 0xA1;                  //freq = resonator/510 = 10 MHz/510
                                //freq = 19.607 kHz
TCCR1B = 0x01;                  //no clock source division

PORTD = PORTD | 0x08;           //Left motor forward  - PORTD[3]=1
PORTD = PORTD & 0xbf;           //Right motor reverse - PORTD[6]=0

                                //Initiate PWM duty cycle variables
                                //Set PWM for left motor to 50
                                //and right motor to 0
OCR1BH = 0x00;
                                //PWM duty cycle CH B left motor
OCR1BL = (unsigned char)(128);
OCR1AH = 0x00;                  //PWM duty cycle CH B right motor
OCR1AL = (unsigned char)(128);
delay(122);                     //delay 1s
OCR1BL = (unsigned char)(0);    //motors off
OCR1AL = (unsigned char)(0);
}

//************************************************************
```

8.5.7 MOUNTAIN MAZE

The mountain maze was constructed from plywood, chicken wire, expandable foam, plaster cloth and Bondo. A rough sketch of the desired maze path was first constructed. Care was taken to ensure the pass was wide enough to accommodate the robot. The maze platform was constructed from 3/8 inch plywood on 2 by 4 inch framing material. Maze walls were also constructed from the plywood and supported with steel L brackets.

With the basic structure complete, the maze walls were covered with chicken wire. The chicken wire was secured to the plywood with staples. The chicken wire was then covered with plaster cloth (Creative Mark Artist Products #15006). To provide additional stability, expandable foam was sprayed under the chicken wire (Guardian Energy Technologies, Inc. Foam It Green 12). The mountain scene was then covered with a layer of Bondo for additional structural stability. Bondo is a two-part putty that hardens into a strong resin. Mountain pass construction steps are illustrated in Figure 8.20. The robot is shown in the maze in Figure 8.21.

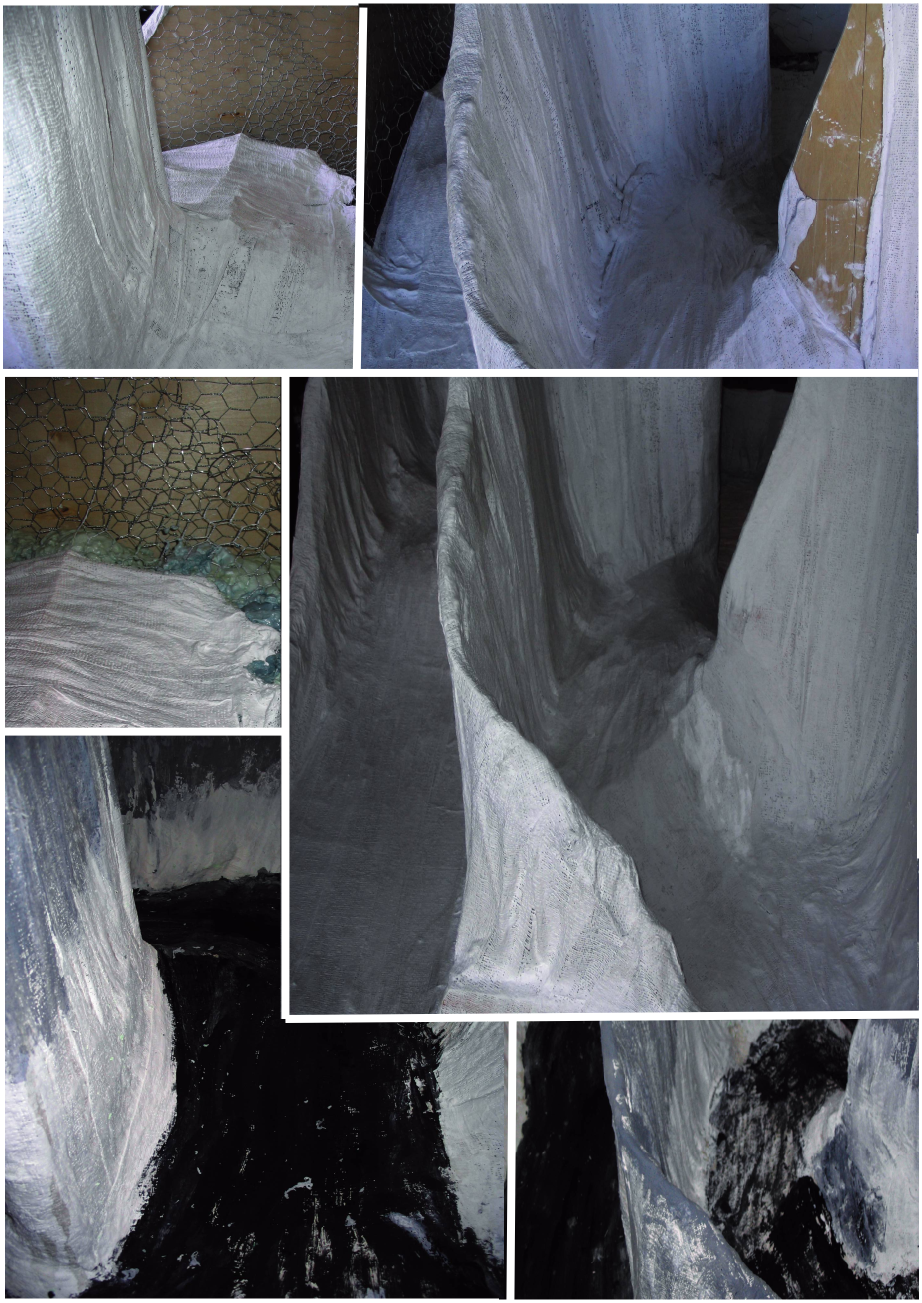


Figure 8.20: Mountain maze.

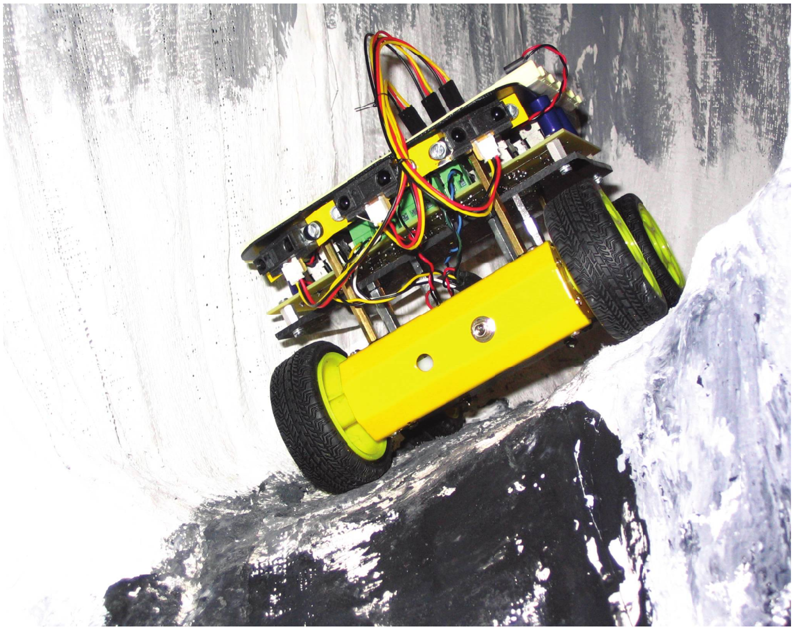


Figure 8.21: Robot in maze.

8.5.8 PROJECT EXTENSIONS

- Modify the PWM turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as variables to the function.

- Equip the motor with another IR sensor that looks down toward the maze floor for "land mines." A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate the maze by moving slowly back and forth for three seconds and flashing a large LED.

- Develop a function for reversing the robot.

- The current design is a two-wheel, front-wheel drive system. Modify the design for a two-wheel, rear-wheel drive system.

- The current design is a two-wheel, front-wheel drive system. Modify the design for a four-wheel drive system.

- Develop a four-wheel drive system which includes a tilt sensor. The robot should increase motor RPM (duty cycle) for positive inclines and reduce motor RPM (duty cycle) for negatives inclines.

8.6 WEATHER STATION

This design was originally provided in *Embedded Systems Design with the Atmel AVR® Microcontroller*. The design is used with permission of Morgan & Claypool. In this project we design a weather station to sense wind direction and to measure ambient temperature. The measured temperature values are displayed on an LCD in Fahrenheit. The wind direction is displayed on LEDs arranged in a circular pattern. The wind direction and temperature are transmitted serially to an external device.

8.6.1 REQUIREMENTS

The requirements for this system include:

- sense wind direction and measure ambient temperature;
- display on an LCD;
- display measured temperature in Fahrenheit on an LCD;
- display wind direction on LEDs arranged in a circular pattern; and
- transmit serially wind direction and temperature data.

8.6.2 STRUCTURE CHART

To begin the design process, a structure chart is used to partition the system into definable subsystems. We employ a top-down design/bottom-up implementation approach. The structure chart for the weather station is shown in Figure 8.22. The three main microcontroller subsystems

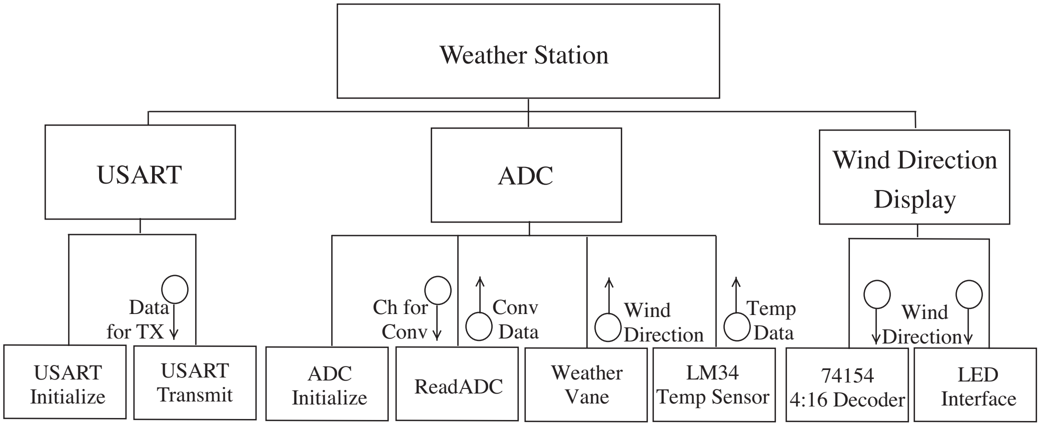


Figure 8.22: Weather station structure chart.

needed for this project are the USART for serial communication, the ADC system to convert the analog voltages from the LM34 temperature sensor and a weather vane into digital signals, and the wind direction display. This display consists of a 74154, a 4-to-16 decoder and 16 individual LEDs to display wind direction. The system is partitioned until the lowest level of the structure chart contains "doable" pieces of hardware components or software functions. Data flow is shown on the structure chart as directed arrows.

8.6.3 CIRCUIT DIAGRAM

The circuit diagram for the weather station is shown in Figure 8.23. The weather station is equipped with two input sensors: the LM34 to measure temperature and the weather vane to measure wind direction. Both of the sensors provide an analog output that is fed to PORTA[0] and PORTA[1]. The LM34 provides 10 mV output per degree Fahrenheit. The weather vane provides 0–5 VDC for 360° of vane rotation. The weather vane must be oriented to a known direction with the output voltage at this direction noted. We assume that 0 VDC corresponds to North and the voltage increases as the vane rotates clockwise to the East. The vane output voltage continues to increase until North is again reached at 5 VDC and then rolls over back to zero volts. All other directions are derived from this reference point.

An LCD is connected to PORTC for data and PORTD[7:6] for the enable and command/data control lines. There are 16 different LEDs for the wind speed indicator. Rather than use 16 microcontroller pins, the binary value of the LED for illumination should be sent to the 74154 4-to-16 decoder. The decoder provides a "one cold" output as determined by the binary code provided on PORTA[7:4]. For example, when A_{16} is sent to the 74154 input, output $/Y10$ is asserted low, while all other outputs remain at logic high. The 74154 is from the standard TTL family. It has sufficient current sink capability ($I_{OL} = 16$ mA) to meet the current requirements of an LED ($V_f = 1.5$ VDC, $I_f = 15$ mA).

8.6.4 UML ACTIVITY DIAGRAMS

The UML activity diagram for the main program is shown in Figure 8.24. After initializing the subsystems, the program enters a continuous loop where temperature and wind direction are sensed and displayed on the LCD and the LED display. The sensed values are then transmitted via the USART. The system then executes a delay routine to configure how often the temperature and wind direction parameters should be updated. We leave the construction of the individual UML activity diagrams for each function as an end of chapter exercise.

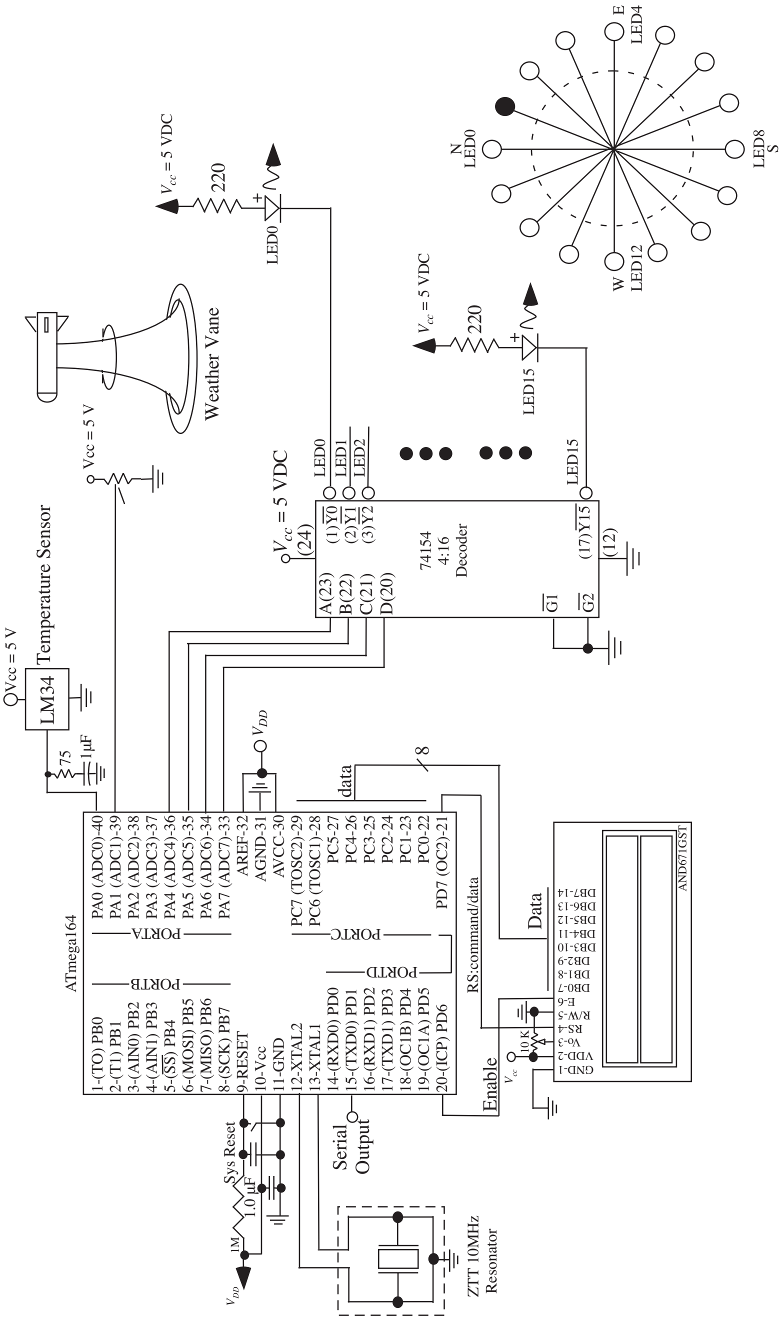


Figure 8.23: Circuit diagram for weather station.

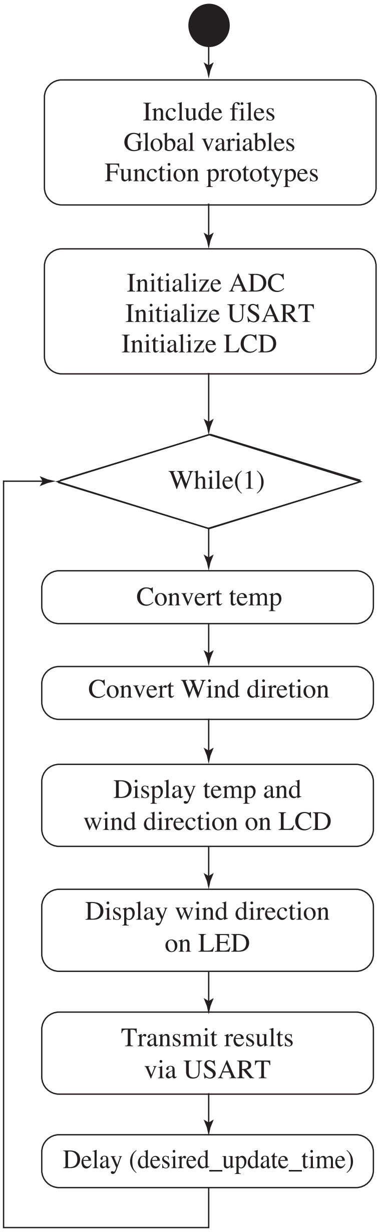


Figure 8.24: Weather station UML activity diagram.

8.6.5 MICROCONTROLLER CODE

```
//*************************************************************
//weather_station.c
//*************************************************************

//include file*************************************************
#include <iom164pv.h>

//function prototypes******************************************
void initialize_ports(void);
void initialize_ADC(void);
void temperature_to_LCD(unsigned int ADCValue);
unsigned int readADC(unsigned char);
void LCD_init(void);
void putChar(unsigned char);
void putcommand(unsigned char);
void display_data(void);
void InitUSART_ch1(void);
void USART_TX_ch1(unsigned char data);
void convert_wind_direction(unsigned int);
void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);
void timer0_interrupt_isr(void);

                                   //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:19



//Global Variables***********************************************
unsigned int  temperature, wind_direction;
unsigned int  binary_weighted_voltage_low;
unsigned int  binary_weighted_voltage_high;
unsigned char dir_tx_data;

void main(void)
{
initialize_ports();
initialize_ADC();
```

```
void InitUSART_ch1();
LCD_init();
init_timer0_ovf_interrupt();

while(1)
  {
  //temperature data: read -> display -> transmit
  temperature = readADC(0x00);       //Read temp from LM34
  temperature_to_LCD (temperature);  //Conv, display temp on LCD
  USART_TX_ch1((unsigned char)(binary_weighted_voltage_low));
  USART_TX_ch1((unsigned char)(binary_weighted_voltage_high >>8));

  //wind direction data: read -> display -> transmit
  wind_direction = readADC(0x01);          //Read wind direction
  convert_wind_direction(wind_direction); //Conv wind dir -> tx
  USART_TX_ch1((unsigned char)(binary_weighted_voltage_low));
  USART_TX_ch1((unsigned char)(binary_weighted_voltage_high >>8));

  //delay 15 minutes
  delay(2307):
  }
}

//*****************************************************************

void initialize_ports()
{
DDRD = 0xFB;
DDRC = 0xFF;
DDRB = 0xFF;
}

//******************************************************************

void initialize_ADC()
{
ADMUX = 0;                          //Select channel 0
                                    //Enable ADC and set module en ADC
ADCSRA = 0xC3;                      //Set module prescalar to 8
```

```
while(!(ADCSRA & 0x10));          //Wait until conversion is ready
ADCSRA |= 0x10;                   //Clear conversion ready flag
}

//*****************************************************************

unsigned int readADC(unsigned char channel)
{
unsigned int binary_weighted_voltage, binary_weighted_voltage_low;
unsigned int binary_weighted_voltage_high;//weighted binary voltage

ADMUX = channel;                       //Select channel
ADCSRA |= 0x43;                        //Start conversion
                                       //Set ADC module prescalar
                                       //to 8 critical for
                                       //accurate ADC results
while (!(ADCSRA & 0x10));              //Check if conversion is ready
ADCSRA |= 0x10;                        //Clr conv rdy flag - set the bit
binary_weighted_voltage_low = ADCL;    //Read 8 low bits first
                                       //Read 2 high bits,
                                       //mult by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage = binary_weighted_voltage_low +
                          binary_weighted_voltage_high;
return binary_weighted_voltage;        //ADCH:ADCL
}

//***************************************************************
//LCD_Init: initialization for AND671GST 1x16 character display LCD
//LCD configured as two 8 character lines in a 1x16 array
//LCD data bus (pin 14-pin7) MICROCHIP ATmega16: PORTC
//LCD RS (pin 4) MICROCHIP ATmega16: PORTD[7]
//LCD E (pin 6) MICROCHIP ATmega16: PORTD[6]
//****************************************************************

void LCD_init(void)
{
delay(1);
delay(1);
delay(1);
                                   //Output command string to
                                   //Initialize LCD
putcommand(0x38);                  //Function set 8-bit
delay(1);
putcommand(0x38);                  //Function set 8-bit
delay(1);
putcommand(0x38);                  //Function set 8-bit
putcommand(0x38);                  //One line, 5x7 char
putcommand(0x0E);                  //Display on
putcommand(0x01);                  //Display clear-1.64 ms
putcommand(0x06);                  //Entry mode set
putcommand(0x00);                  //Clear disp, cursor at home
putcommand(0x00);                  //Clear disp, cursor at home
}


//**************************************************************

void putChar(unsigned char c)
{
DDRC = 0xff;                       //Set PORTC as output
DDRD = DDRD|0xC0;                  //Make PORTD[7:6] output
PORTC = c;
PORTD = PORTD|0x80;                //RS=1
PORTD = PORTD|0x40;                //E=1
PORTD = PORTD&0xbf;                //E=0
delay(1);
}

//***************************************************************

void putcommand(unsigned char d)
{
DDRC = 0xff;                        //Set PORTC as output
DDRD = DDRD|0xC0;                   //Make PORTD[7:6] output
PORTD = PORTD&0x7f;                 //RS=0
PORTC = d;
PORTD = PORTD|0x40;                 //E=1
PORTD = PORTD&0xbf;                  //E=0
delay(1);
}

//*****************************************************************

void temperature_to_LCD(unsigned int ADCValue)
{
float voltage,temperature;
unsigned int tens, ones, tenths;

voltage = (float)ADCValue*5.0/1024.0;

temperature = voltage*100;


tens = (unsigned int)(temperature/10);
ones = (unsigned int)(temperature-(float)tens*10);
tenths = (unsigned int)(((temperature-(float)tens*10)-(float)ones)*10);

putcommand(0x01);                   //Cursor home
putcommand(0x80);                   //DD RAM location 1  - line 1
putChar((unsigned char)(tens)+48);
putChar((unsigned char)(ones)+48);
putChar('.');
putChar((unsigned char)(tenths)+48);
putChar('F');
}

//*****************************************************************

void convert_wind_direction(unsigned int wind_dir_int)
{
float wind_dir_float;
                                  //Convert wind direction to float
wind_dir_float = ((float)wind_dir_int)/1024.0) * 5;

//N - LED0
if((wind_dir_float <= 0.15625)||(wind_dir_float > 4.84375))
```

```
   {
   putcommand(0x01);                //Cursor to home
   putcommand(0xc0);                //DD RAM location 1 - line 2
   putchar('N');                    //LCD displays: N
   PORTA = 0x00;                    //Illuminate LED 0
   }

//NNE - LED1
if((wind_dir_float > 0.15625)||(wind_dir_float <= 0.46875))
   {
   putcommand(0x01);               //Cursor to home
   putcommand(0xc0);               //DD RAM location 1 - line 2
   putchar('N');                   //LCD displays: NNE
   putchar('N');
   putchar('E');
   PORTA = 0x10;                   //Illuminate LED 1
   }

//NE - LED2
if((wind_dir_float > 0.46875)||(wind_dir_float <= 0.78125))
   {
   putcommand(0x01);               //Cursor to home
   putcommand(0xc0);               //DD RAM location 1 - line 2
   putchar('N');                   //LCD displays: NE
   putchar('E');
   PORTA = 0x20;                   //Illuminate LED 2
   }

//ENE - LED3
if((wind_dir_float > 0.78125)||(wind_dir_float <= 1.09375))
   {
   putcommand(0x01);                //Cursor to home
   putcommand(0xc0);                //DD RAM location 1 - line 2
   putchar('E');                    //LCD displays: NNE
   putchar('N');
   putchar('E');
   PORTA = 0x30;                    //Illuminate LED 3
   }
```

```
//E - LED4
if((wind_dir_float > 1.09375)||(wind_dir_float <= 1.40625))
   {
   putcommand(0x01);               //Cursor to home
   putcommand(0xc0);               //DD RAM location 1 - line 2
   putchar('E');                   //LCD displays: E
   PORTA = 0x40;                   //Illuminate LED 4
   }

//ESE - LED5
if((wind_dir_float > 1.40625)||(wind_dir_float <= 1.71875))
   {
   putcommand(0x01);               //Cursor to home
   putcommand(0xc0);               //DD RAM location 1 - line 2
   putchar('E');                   //LCD displays: ESE
   putchar('S');
   putchar('E');
   PORTA = 0x50;                   //Illuminate LED 5
   }

//SE - LED6
if((wind_dir_float > 1.71875)||(wind_dir_float <= 2.03125))
   {
   putcommand(0x01);               //Cursor to home
   putcommand(0xc0);                //DD RAM location 1 - line 2
   putchar('S');                    //LCD displays: SE
   putchar('E');
   PORTA = 0x60;                    //Illuminate LED 6
   }

//SSE - LED7
if((wind_dir_float > 2.03125)||(wind_dir_float <= 2.34875))
   {
   putcommand(0x01);                //Cursor to home
   putcommand(0xc0);                //DD RAM location 1 - line 2
   putchar('S');                    //LCD displays: SSE
   putchar('S');
   putchar('E');
   PORTA = 0x70;                    //Illuminate LED 7
   }

//S - LED8
if((wind_dir_float > 2.34875)||(wind_dir_float <= 2.65625))
   {
   putcommand(0x01);                //Cursor to home
   putcommand(0xc0);                //DD RAM location 1 - line 2
   putchar('S');                    //LCD displays: S

   PORTA = 0x80;                    //Illuminate LED 8
   }

//SSW - LED9
if((wind_dir_float > 2.65625)||(wind_dir_float <= 2.96875))
   {
   putcommand(0x01);                //Cursor to home
   putcommand(0xc0);                //DD RAM location 1 - line 2
   putchar('S');                    //LCD displays: SSW
   putchar('S');
   putchar('W');
   PORTA = 0x90;                    //Illuminate LED 9
   }

//SW - LED10 (A)
if((wind_dir_float > 2.96875)||(wind_dir_float <= 3.28125))
   {
   putcommand(0x01);                //Cursor to home
   putcommand(0xc0);                //DD RAM location 1 - line 2
   putchar('S');                    //LCD displays: SW
   putchar('W');
   PORTA = 0xa0;                    //Illuminate LED 10 (A)
   }

//WSW - LED11 (B)
if((wind_dir_float > 3.28125)||(wind_dir_float <= 3.59375))
   {
   putcommand(0x01);                //Cursor to home
   putcommand(0xc0);                //DD RAM location 1 - line 2
   putchar('W');                    //LCD displays: WSW
   putchar('S');
   putchar('W');
   PORTA = 0xb0;                     //Illuminate LED 11 (B)
   }

//W - LED12 (C)
if((wind_dir_float > 3.59375)||(wind_dir_float <= 3.90625))
   {
   putcommand(0x01);                 //Cursor to home
   putcommand(0xc0);                 //DD RAM location 1 - line 2
   putchar('W');                     //LCD displays: W
   PORTA = 0xc0;                     //Illuminate LED 12 (C)
   }

//WNW - LED13 (D)
if((wind_dir_float > 3.90625)||(wind_dir_float <= 4.21875))
   {
   putcommand(0x01);                 //Cursor to home
   putcommand(0xc0);                 //DD RAM location 1 - line 2
   putchar('W');                     //LCD displays: WNW
   putchar('N');
   putchar('W');
   PORTA = 0xd0;                     //Illuminate LED 13 (D)
   }

//NW - LED14 (E)
if((wind_dir_float > 4.21875)||(wind_dir_float <= 4.53125))
   {
   putcommand(0x01);                 //Cursor to home
   putcommand(0xc0);                 //DD RAM location 1 - line 2
   putchar('N');                     //LCD displays: NW
   putchar('W');
   PORTA = 0xe0;                     //Illuminate LED 14 (E)
   }

//NNW - LED15(F)
if((wind_dir_float > 4.53125)||(wind_dir_float < 4.84375))
   {
   putcommand(0x01);                 //Cursor to home
   putcommand(0xc0);                   //DD RAM location 1 - line 2
   putchar('N');                       //LCD displays: NNW
   putchar('N');
   putchar('W');
   PORTA = 0xf0;                       //Illuminate LED 15 (F)
   }
}

//*************************************************************

void InitUSART_ch1(void)
{
//USART Channel 1 initialization
//System operating frequency: 10 MHz
// Comm Parameters: 8 bit Data, 1 stop, No Parity
// USART Receiver: Off
// USART Trasmitter: On
// USART Mode: Asynchronous
// USART Baud Rate: 9600

UCSR1A=0x00;
UCSR1B=0x18;                           //RX on, TX on
UCSR1C=0x06;                           //1 stop bit, No parity
UBRR1H=0x00;
UBRR1L=0x40;
}

//*************************************************************

void USART_TX_ch1(unsigned char data)
{
                                       //Set USART Data Register
                                       //data register empty?
while(!(UCSR1A & 0x20));
 UDR1= data;                           //Sets value in ADCH to
                                       //value in the USART
                                       //Data Register
}
```

```
//***************************************************************

unsigned char USART_RX_ch1(void)
{
unsigned char rx_data;
                                      //receive is complete?
while(!(UCSR1A & 0x80));
rx_data=UDR1;                         //Returns data
return rx_data;
}

//****************************************************************
//int_timer0_ovf_interrupt(): The Timer0 overflow interrupt is
//being  employed as a time base for a master timer for this
//project.  The ceramic  resonator operating at 10 MHz is divided
//by 256.  The 8-bit Timer0 register (TCNT0) overflows every 256
//counts or every 6.55 ms.
//****************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04; //div timer0 timebase by 256, overflow every 6.55ms
TIMSK0 = 0x01; //enable timer0 overflow interrupt
asm("SEI");    //enable global interrupt
}

//**************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing
//the corresponding interrupt handling vector.
//**************************************************************


void timer0_interrupt_isr(void)
{
input_delay++;                         //input delay processing
}

//**************************************************************
```

```
//delay(unsigned int num_of_6_55ms_interrupts): This generic
//delay function provides the specified delay as the number of
//6.55 ms "clock ticks" from the Timer0 interrupt.
//Note: This function is only valid when using a 10 MHz crystal
//or ceramic resonator.
//*************************************************************

void delay(unsigned int number_of_6_55ms_interrupts)
{
TCNT0 = 0x00;                          //reset timer0
input_delay = 0;
while(input_delay <= number_of_6_55ms_interrupts)
  {
  ;
  }
}

//**************************************************************
```

8.7 MOTOR SPEED CONTROL

This design was originally provided in *Embedded Systems Design with the Atmel AVR® Microcontroller*. The design is used with permission of Morgan & Claypool.

In this project, we will control the speed of a 24 VDC, 1500 RPM motor using an external potentiometer. The motor is equipped with an optical tachometer, which produces three channels of information. Two of the channels output quadrature related (90° out of phase with one another) 0.5 V peak sinusoidal signals. The third channel provides a single pulse index signal for every motor revolution, as shown in Figure 8.25 [Brother].

8.8 CIRCUIT DIAGRAM

We employ the pulse width modulation (PWM) system of the Microchip ATmega164 to set the motor speed as determined by the potentiometer setting. A potentiometer setting of 0 VDC equates to a 50% duty cycle; whereas, a 5 VDC setting corresponds to a 100% duty cycle. The motor speed and duty cycle will be displayed on an LCD as shown in Figure 8.26.

In this application the PWM baseline frequency may be set to a specific frequency. The duty cycle will be varied to adjust the effective voltage delivered to the motor. For example, a 50% duty cycle will deliver an effective value of 50% of the DC motor supply voltage to the motor.

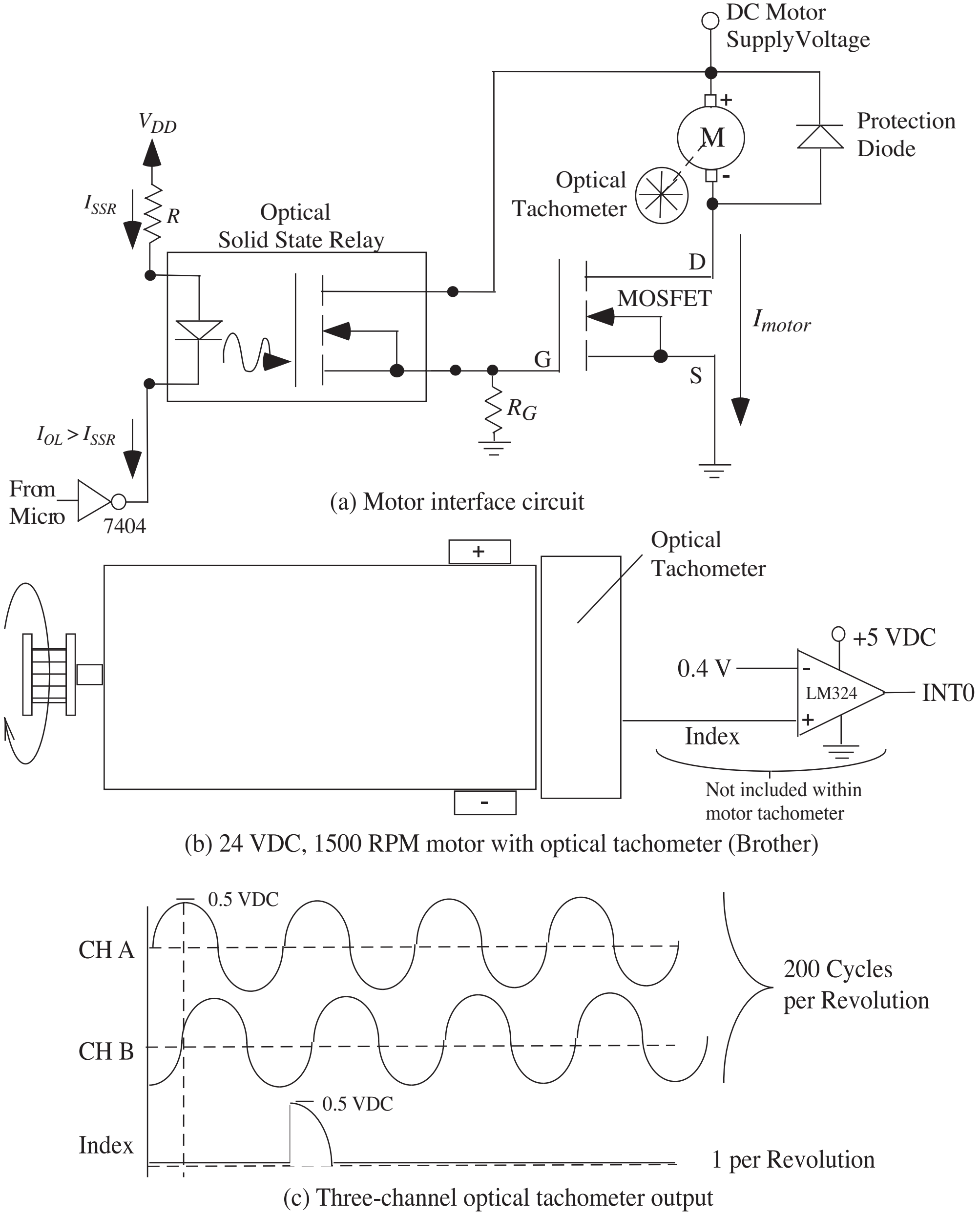


Figure 8.25: 24 VDC, 1500 RPM motor equipped with a three-channel optical encoder.

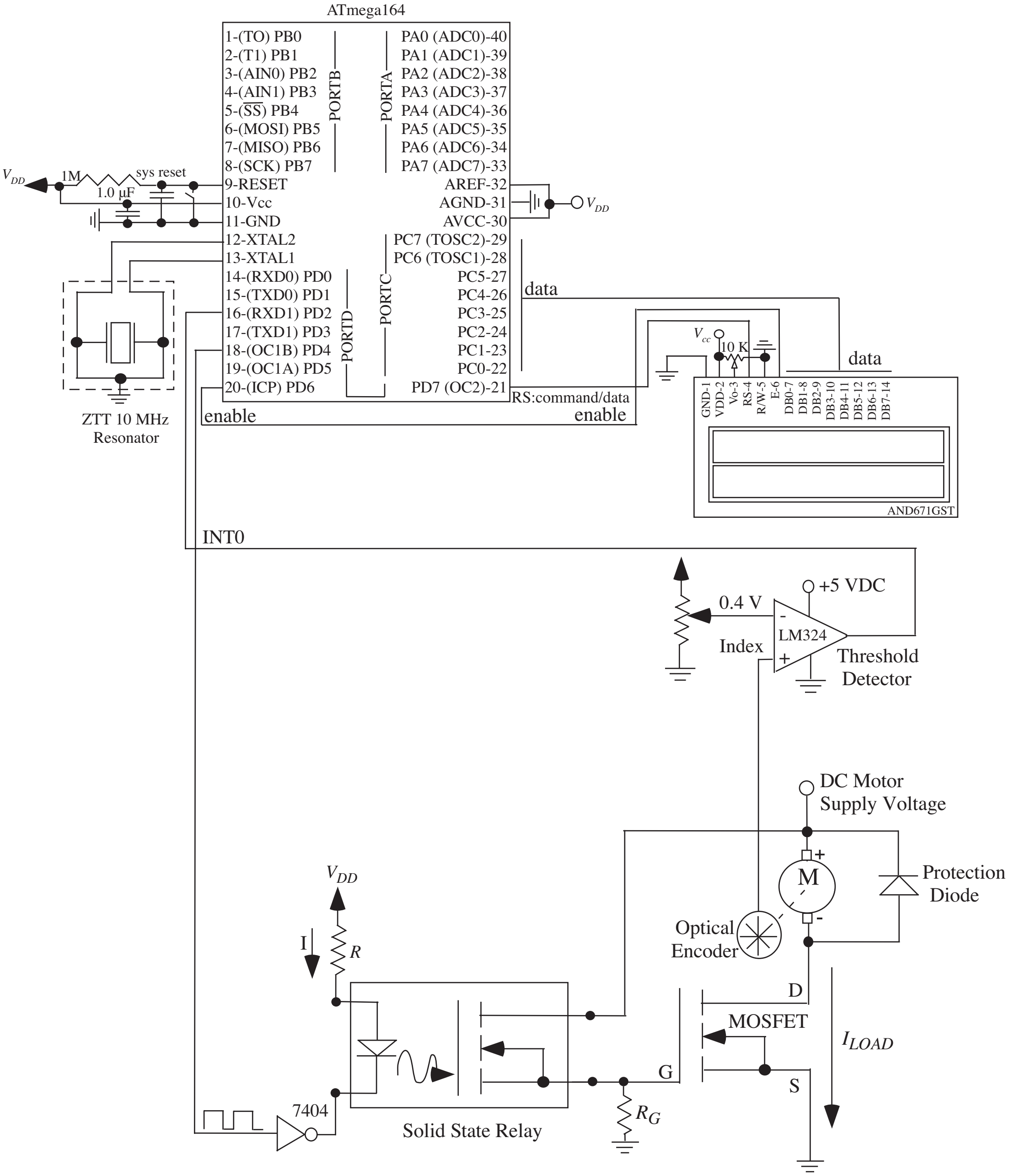


Figure 8.26: Circuit diagram for a 24 VDC, 1500 RPM motor equipped with a three-channel optical encoder.

The microcontroller is not directly connected to the motor. The PWM control signal from OC1B (pin 18) is fed to the motor through an optical solid state relay (SSR), as shown in Figure 8.26. This isolates the microcontroller from the noise of the motor. The output signal from SSR is fed to the MOSFET which converts the low-level control signal to voltage and current levels required by the motor.

Motor speed is monitored via the optical encoder connected to the motor shaft. The index output of the motor provides a pulse for each rotation of the motor. The signal is converted to a TTL compatible signal via the LM324 threshold detector. The output from this stage is fed back to INTO to trigger an external interrupt. An interrupt service routine captures the time since the last interrupt. This information is used to speed up or slow down the motor to maintain a constant speed.

8.8.1 REQUIREMENTS

- Generate a 1 kHz PWM signal.
- Vary the duty cycle from 50–100%, which is set by the potentiometer, i.e., a 50% duty cycle corresponds to 0 VDC and a 90% duty cycle is equal to 5 VDC.
- Display the motor RPM and duty cycle on an AND671GST LCD.
- Load the motor and perform compensation to return the RPMs to original value.

8.8.2 STRUCTURE CHART

The structure chart for the motor speed control project is shown in Figure 8.27.

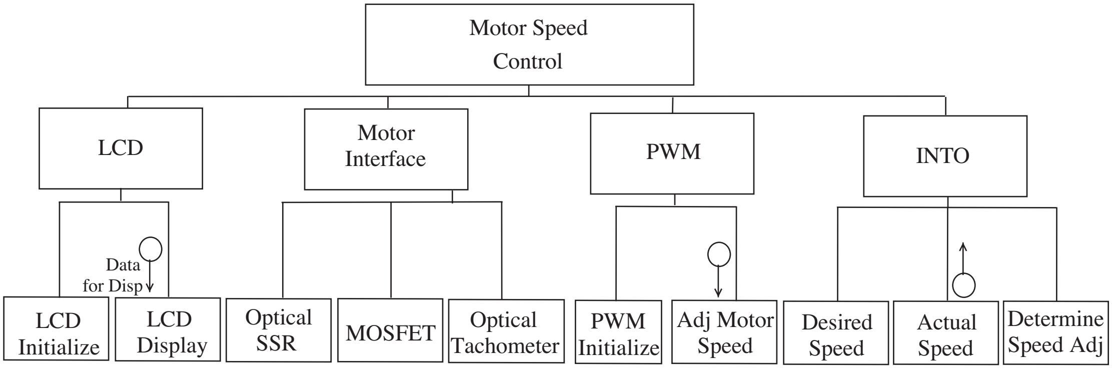


Figure 8.27: Structure chart for the motor speed control project.

8.8.3 UML ACTIVITY DIAGRAMS

The UML activity diagrams for the motor speed control project are shown in Figure 8.28.

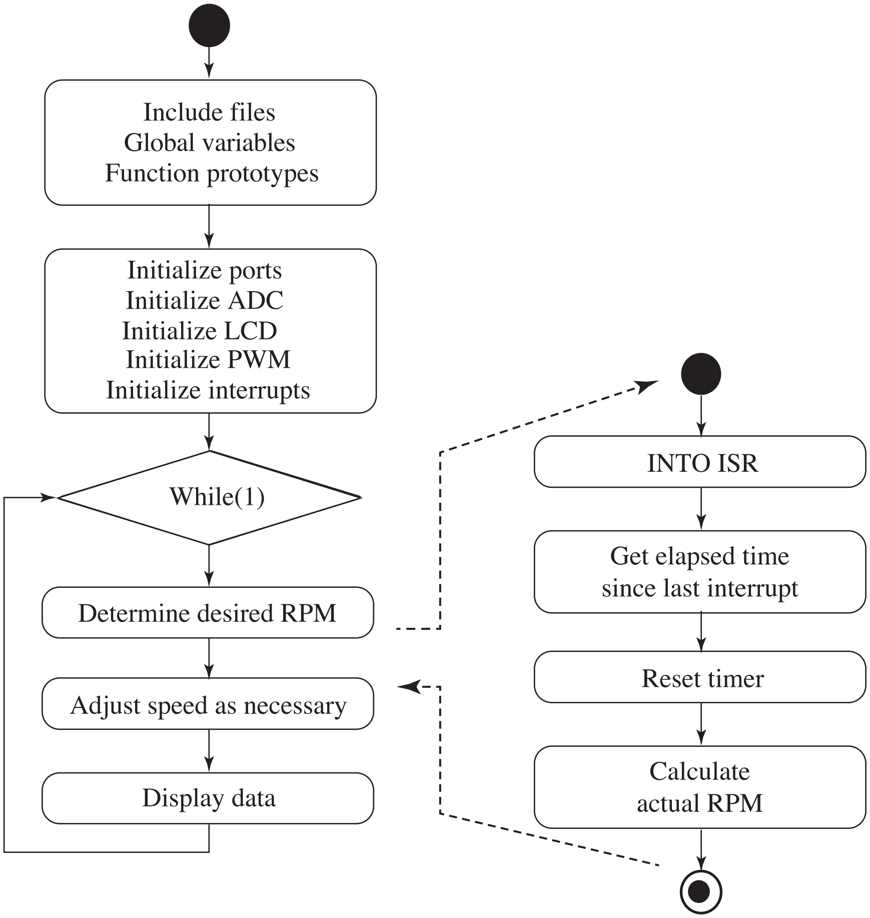


Figure 8.28: UML activity diagrams for the motor speed control project.

8.8.4 MICROCONTROLLER CODE

The code for the motor speed control project follows. Note that this code was implemented using the gcc AVR compiler. Note the different notation used for including the header file and configuring interrupts.

```
//*************************************************************
//Code written by Geoff Luke, MSEE
//Last Updated: April 30, 2017
//*************************************************************
//
//Note: This program was written using the gcc AVR compiler
```

```
//
//Description: This program powers a motor to run at 1125 to
//2025 rpm.  The value is set by a potentiometer.  The
//microcontroller receives feedback from an optical tachometer
//that triggers external interrupt 0.  The desired speed is
//maintained even when the motor is loaded.
//
//Port connection:
//  Port C 0-7: used as data output to LCD
//  Port D 6-7: control pins for LCD
//  Port D 2:   External interrupt 0 (from tachometer)
//  Port A 0:   Used as ATD input
//  Port B 0:   PWM output
//
//**************************************************************

//include files**************************************************
#include <avr\io.h>
#include <avr\interrupt.h>

//function prototypes*******************************************
void initialize_ports();
void initialize_ADC();
unsigned int readADC(unsigned char);
void LCD_init();
void putChar(unsigned char);
void putcommand(unsigned char);
void PWM_init(void);
void display_data(void);
void delay_5ms();

//global variables***********************************************
unsigned int actualRPM, desiredRPM;

ISR(INT0_vect)
{
unsigned int time = TCNT0;
TCNT0 = 0x00;
actualRPM = 3906/time*60;
}

//*****************************************************************
int main(void)
{
initializePorts();
initializeADC();
LCD_init();
PWM_init();

MCUCR = 0x02;
GICR = 0x40;
TCCR0 = 0x05;
actualRPM = 0;
sei();

while(1)
  {
  desiredRPM = (unsigned int)(0.878906*(float)readADC(0)+1125.0);
  if(desiredRPM > actualRPM && OCR1BL != 0xFF)
    {
    OCR1BL = OCR1BL+1;
    }
  else if(desiredRPM < actualRPM && OCR1BL != 0x00)
    {
    OCR1BL = OCR1BL-1;
    }
  else
    {
    OCR1BL = OCR1BL;
    }
  displayData();
  }
return 0;
}

//***************************************************************

void initialize_ports()
```

```
{
DDRD = 0xFB;
DDRC = 0xFF;
DDRB = 0xFF;
}

//***************************************************************

void initialize_ports()
{
DDRD = 0xFB;
DDRC = 0xFF;
DDRB = 0xFF;
}

//***************************************************************

void initialize_ADC()
{
ADMUX = 0;                      //Select channel 0
                                //En ADC and set module en ADC
ADCSRA = 0xC3;                  //Set module prescalar to 8
while(!(ADCSRA & 0x10));        //Wait until conversion is ready
ADCSRA |= 0x10;                 //Clear conversion ready flag
}

//****************************************************************

unsigned int readADC(unsigned char channel)
{
unsigned int binary_weighted_voltage, binary_weighted_voltage_low;
unsigned int binary_weighted_voltage_high;//weighted binary voltage

ADMUX = channel;                    //Select channel
ADCSRA |= 0x43;                     //Start conversion
                                    //Set ADC module prescalar
                                    //to 8 critical for
                                    //accurate ADC results
while (!(ADCSRA & 0x10));           //Check if conversion is ready
```

```
ADCSRA |= 0x10;                     //Clear conv rdy flag
binary_weighted_voltage_low = ADCL; //Read 8 low bits first
                                    //Read 2 high bits,
                                    //multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage = binary_weighted_voltage_low +
                          binary_weighted_voltage_high;
return binary_weighted_voltage;     //ADCH:ADCL
}

//**************************************************************
//LCD_Init: init for an LCD connected in the following manner:
//LCD: AND671GST 1x16 character display
//LCD configured as two 8 character lines in a 1x16 array
//LCD data bus (pin 14-pin7) MICROCHIP ATmega16: PORTC
//LCD RS (pin 4) MICROCHIP ATmega16: PORTD[7]
//LCD E (pin 6) MICROCHIP ATmega16: PORTD[6]
//***************************************************************

void LCD_init(void)


{
delay(1);
delay(1);
delay(1);
                                 //Output command string to
                                 //Initialize LCD
putcommand(0x38);                //Function set 8-bit
delay(1);
putcommand(0x38);                //Function set 8-bit
delay(1);
putcommand(0x38);                //Function set 8-bit
putcommand(0x38);                //One line, 5x7 char
putcommand(0x0E);                //Display on
putcommand(0x01);                //Display clear-1.64 ms
putcommand(0x06);                //Entry mode set
putcommand(0x00);                //Clear display, cursor at home
putcommand(0x00);                //Clear display, cursor at home
```

```
}

//*****************************************************************

void putChar(unsigned char c)
{
DDRC = 0xff;                        //Set PORTC as output
DDRD = DDRD|0xC0;                   //Make PORTD[7:6] output
PORTC = c;
PORTD = PORTD|0x80;                 //RS=1
PORTD = PORTD|0x40;                 //E=1
PORTD = PORTD&0xbf;                 //E=0
delay(1);
}


//*****************************************************************

void putcommand(unsigned char d)
{
DDRC = 0xff;                         //Set PORTC as output
DDRD = DDRD|0xC0;                    //Make PORTD[7:6] output
PORTD = PORTD&0x7f;                  //RS=0
PORTC = d;
PORTD = PORTD|0x40;                  //E=1
PORTD = PORTD&0xbf;                  //E=0
delay(1);
}

//******************************************************************
void display_data(void)
{
unsigned int thousands, hundreds, tens, ones, dutyCycle;

thousands = desiredRPM/1000;
hundreds = (desiredRPM - 1000*thousands)/100;
tens = (desiredRPM - 1000*thousands - 100*hundreds)/10;
ones = (desiredRPM - 1000*thousands - 100*hundreds - 10*tens);
```

```
putcommand(0x80);
putChar((unsigned char)(thousands)+48);
putChar((unsigned char)(hundreds)+48);
putChar((unsigned char)(tens)+48);

putChar((unsigned char)(ones)+48);
putChar('R');
putChar('P');
putChar('M');
putcommand(0xC0);

thousands = actualRPM/1000;
hundreds = (actualRPM - 1000*thousands)/100;
tens = (actualRPM - 1000*thousands - 100*hundreds)/10;
ones = (actualRPM - 1000*thousands - 100*hundreds - 10*tens);

putcommand(0xC0);
putChar((unsigned char)(thousands)+48);
putChar((unsigned char)(hundreds)+48);
putChar((unsigned char)(tens)+48);
putChar((unsigned char)(ones)+48);
putChar('R');
putChar('P');
putChar('M');

putChar(' ');
putChar(' ');

dutyCycle = OCR1BL*100/255;

hundreds = (dutyCycle)/100;
tens = (dutyCycle - 100*hundreds)/10;
ones = (dutyCycle - 100*hundreds - 10*tens);

if(hundreds > 0)
  {
  putChar((unsigned char)(hundreds)+48);
  }
```

```
else

  {
  putChar(' ');
  }
putChar((unsigned char)(tens)+48);
putChar((unsigned char)(ones)+48);
putChar('
}

//*************************************************************

void PWM_init(void)
{
unsigned int Open_Speed_int;
float Open_Speed_float;
int PWM_duty_cycle;


Open_Speed_int = readADC(0x02);           //Open Speed Setting
                                          //unsigned int
                                          //Convert to max duty
                                          //Cycle setting 0 VDC =
                                          //50
                                          //100
Open_Speed_float = ((float)(Open_Speed_int)/(float)(0x0400));
                                          //Convert volt to
                                          //PWM constant 127-255
Open_Speed_int = (unsigned int)((Open_Speed_float * 127) + 128.0);
                                          //Configure PWM clock
TCCR1A = 0xA1;                            //freq = resonator/510
                                          // = 4 MHz/510
                                          //freq = 19.607 kHz
TCCR1B = 0x02;                            //Clock source
                                          //division of 8
                                          //Initiate PWM duty cycle
                                          //variables
PWM_duty_cycle = 255;
OCR1BH = 0x00;
OCR1BL = (unsigned char)(PWM_duty_cycle); //set PWM duty cycle CH
}

//*************************************************************
```

8.9 SUMMARY

In this chapter, we discussed the embedded system design process and related tools. We also applied the process to real-world designs. It is essential to follow a systematic, disciplined approach to embedded systems design to successfully develop a prototype that meets established requirements.

8.10 REFERENCES AND FURTHER READING

Anderson, M. Help wanted: Embedded engineers why the United States is losing its edge in embedded systems. *IEEE—USA Today's Engineer*, February 2008. 245

Barrett, S. F. and Pack, D. J. *Atmel AVR® Microcontroller Primer Programming and Interfacing*. Morgan & Claypool Publishers. San Rafael, CA, 2010. DOI: 10.2200/S00100ED1V01Y200712DCS015

Barrett, S. F. *Embedded Systems Design with the Atmel AVR® Microcontroller*. Morgan & Claypool Publishers, San Rafael, CA, 2010. DOI: 10.2200/S00225ED1V01Y200910DCS025

Barrett, S. F. and Pack, D. J. *Microcontrollers Fundamentals for Engineers and Scientists*. Morgan & Claypool Publishers, San Rafael, CA, 2006. DOI: 10.2200/S00025ED1V01Y200605DCS001 248

Dale, N. and Lilly, S. C. *Pascal Plus Data Structures*, 4th ed., Jones and Bartlett, Englewood Cliffs, NJ, 1995. 250

Fowler, M. and Scott, K. *UML Distilled—A Brief Guide to the Standard Object Modeling Language*, 2nd ed., Addison-Wesley, Boston, 2000. 249

8.11 CHAPTER PROBLEMS

1. What is an embedded system?

2. What aspects must be considered in the design of an embedded system?

3. What is the purpose of the structure chart, UML activity diagram, and circuit diagram?

4. Why is a system design only as good as the test plan that supports it?

5. During the testing process, when an error is found and corrected, what should now be accomplished?

6. Discuss the top-down design, bottom-up implementation concept.

7. Describe the value of accurate documentation.

8. What is required to fully document an embedded systems design?

9. Update the robot action truth table if the robot was equipped with four IR sensors.

10. For the Dagu Magician robot, modify the PWM turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as variables to the function.

11. For the Dagu Magician robot, equip the motor with another IR sensor that looks down for "land mines." A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate it by rotating about its center axis three times and flashing a large LED while rotating.

12. For the Dagu Magician robot, develop a function for reversing the robot.

13. For the 4WD robot, modify the PWM turning commands such that the PWM duty cycle and the length of time the motors are on are sent in as variables to the function.

14. For the 4WD robot, equip the motor with another IR sensor that looks down for "land mines." A land mine consists of a paper strip placed in the maze floor that obstructs a portion of the maze. If a land mine is detected, the robot must deactivate it by rotating about its center axis three times and flashing a large LED while rotating.

15. For the 4WD robot, develop a function for reversing the robot.

16. For the 4WD robot, the current design is a two-wheel, front-wheel drive system. Modify the design for a two-wheel, rear-wheel drive system.

17. For the 4WD robot, the current design is a two-wheel, front-wheel drive system. Modify the design for a four-wheel drive system.

18. For the 4WD robot, develop a four-wheel drive system which includes a tilt sensor. The robot should increase motor RPM (duty cycle) for positive inclines and reduce motor RPM (duty cycle) for negatives inclines.

APPENDIX A

ATmega164 Header File

During C programming, the contents of a specific register may be referred to by name when an appropriate header file is included within your program. The header file provides the link between the register name used within a program and the hardware location of the register.

Provided below is the ATmega164 header file from the ICC AVR compiler. This header file was provided courtesy of ImageCraft (www.iamgecraft.com).

```
#ifndef ___iom164to644pv_h
#define ___iom164to644pv_h

/* ATmega164..644P io header file for
 * ImageCraft ICCAVR compiler
 */

/* 2006/10/01 created as iom644pv.h
   2008/04/21 added USART bits for SPI mode
   2008/05/26 fixed PRR
   2009/02/14 added BODS bits to MCUCR
   2009/06/26 fixed first line #ifdef
*/

/* Port D */
#define PIND (*(volatile unsigned char *)0x29)
#define DDRD (*(volatile unsigned char *)0x2A)
#define PORTD (*(volatile unsigned char *)0x2B)

/* Port C */
#define PINC (*(volatile unsigned char *)0x26)
#define DDRC (*(volatile unsigned char *)0x27)
#define PORTC (*(volatile unsigned char *)0x28)

/* Port B */
#define PINB (*(volatile unsigned char *)0x23)
#define DDRB (*(volatile unsigned char *)0x24)
```

```
#define PORTB (*(volatile unsigned char *)0x25)

/* Port A */
#define PINA (*(volatile unsigned char *)0x20)
#define DDRA (*(volatile unsigned char *)0x21)
#define PORTA (*(volatile unsigned char *)0x22)

/* Timer/Counter Interrupts */
#define TIFR0 (*(volatile unsigned char *)0x35)
#define  OCF0B    2
#define  OCF0A    1
#define  TOV0     0
#define TIMSK0 (*(volatile unsigned char *)0x6E)
#define  OCIE0B   2
#define  OCIE0A   1
#define  TOIE0    0
#define TIFR1 (*(volatile unsigned char *)0x36)
#define  ICF1     5
#define  OCF1B    2
#define  OCF1A    1
#define  TOV1     0
#define TIMSK1 (*(volatile unsigned char *)0x6F)
#define  ICIE1    5
#define  OCIE1B   2
#define  OCIE1A   1
#define  TOIE1    0
#define TIFR2 (*(volatile unsigned char *)0x37)
#define  OCF2B    2
#define  OCF2A    1
#define  TOV2     0
#define TIMSK2 (*(volatile unsigned char *)0x70)
#define  OCIE2B   2
#define  OCIE2A   1
#define  TOIE2    0

/* External Interrupts */
#define EIFR (*(volatile unsigned char *)0x3C)
#define  INTF2    2
#define  INTF1    1
```

```
#define  INTF0    0
#define EIMSK (*(volatile unsigned char *)0x3D)
#define  INT2     2
#define  INT1     1
#define  INT0     0
#define EICRA (*(volatile unsigned char *)0x69)
#define  ISC21    5
#define  ISC20    4
#define  ISC11    3
#define  ISC10    2
#define  ISC01    1
#define  ISC00    0

/* Pin Change Interrupts */
#define PCIFR (*(volatile unsigned char *)0x3B)
#define  PCIF3    3
#define  PCIF2    2
#define  PCIF1    1
#define  PCIF0    0
#define PCICR (*(volatile unsigned char *)0x68)
#define  PCIE3    3
#define  PCIE2    2
#define  PCIE1    1
#define  PCIE0    0
#define PCMSK0 (*(volatile unsigned char *)0x6B)
#define PCMSK1 (*(volatile unsigned char *)0x6C)
#define PCMSK2 (*(volatile unsigned char *)0x6D)
#define PCMSK3 (*(volatile unsigned char *)0x73)

/* GPIOR */
#define GPIOR0 (*(volatile unsigned char *)0x3E)
#define GPIOR1 (*(volatile unsigned char *)0x4A)
#define GPIOR2 (*(volatile unsigned char *)0x4B)

/* EEPROM */
#define EECR (*(volatile unsigned char *)0x3F)
#define  EEPM1    5
#define  EEPM0    4
#define  EERIE    3
```

```
#define  EEMPE    2
#define  EEMWE    2
#define  EEPE     1
#define  EEWE     1
#define  EERE     0
#define EEDR (*(volatile unsigned char *)0x40)
#define EEAR (*(volatile unsigned int *)0x41)
#define EEARL (*(volatile unsigned char *)0x41)
#define EEARH (*(volatile unsigned char *)0x42)

/* GTCCR */
#define GTCCR (*(volatile unsigned char *)0x43)
#define  TSM      7
#define  PSRASY   1
#define  PSR2     1
#define  PSRSYNC  0
#define  PSR10    0

/* Timer/Counter 0 */
#define OCR0B (*(volatile unsigned char *)0x48)
#define OCR0A (*(volatile unsigned char *)0x47)
#define TCNT0 (*(volatile unsigned char *)0x46)
#define TCCR0B (*(volatile unsigned char *)0x45)
#define  FOC0A    7
#define  FOC0B    6
#define  WGM02    3
#define  CS02     2
#define  CS01     1
#define  CS00     0
#define TCCR0A (*(volatile unsigned char *)0x44)
#define  COM0A1   7
#define  COM0A0   6
#define  COM0B1   5
#define  COM0B0   4
#define  WGM01    1
#define  WGM00    0

/* SPI */
#define SPCR (*(volatile unsigned char *)0x4C)
```

```
#define  SPIE     7
#define  SPE      6
#define  DORD     5
#define  MSTR     4
#define  CPOL     3
#define  CPHA     2
#define  SPR1     1
#define  SPR0     0
#define SPSR (*(volatile unsigned char *)0x4D)
#define  SPIF     7
#define  WCOL     6
#define  SPI2X    0
#define SPDR (*(volatile unsigned char *)0x4E)

/* Analog Comparator Control and Status Register */
#define ACSR (*(volatile unsigned char *)0x50)
#define  ACD      7
#define  ACBG     6
#define  ACO      5
#define  ACI      4
#define  ACIE     3
#define  ACIC     2
#define  ACIS1    1
#define  ACIS0    0

/* OCDR */
#define OCDR (*(volatile unsigned char *)0x51)
#define  IDRD     7

/* MCU */
#define MCUSR (*(volatile unsigned char *)0x54)
#define  JTRF     4
#define  WDRF     3
#define  BORF     2
#define  EXTRF    1
#define  PORF     0
#define MCUCR (*(volatile unsigned char *)0x55)
#define  JTD      7
#define  BODS     6
```

```
#define  BODSE    5
#define  PUD      4
#define  IVSEL    1
#define  IVCE     0

#define SMCR (*(volatile unsigned char *)0x53)
#define  SM2      3
#define  SM1      2
#define  SM0      1
#define  SE       0

/* SPM Control and Status Register */
#define SPMCSR (*(volatile unsigned char *)0x57)
#define  SPMIE    7
#define  RWWSB    6
#define  SIGRD    5
#define  RWWSRE   4
#define  BLBSET   3
#define  PGWRT    2
#define  PGERS    1
#define  SPMEN    0

/* RAMPZ */
/* #define RAMPZ  (*(volatile unsigned int *)0x5B) */

/* Stack Pointer */
#define SP   (*(volatile unsigned int *)0x5D)
#define SPL  (*(volatile unsigned char *)0x5D)
#define SPH  (*(volatile unsigned char *)0x5E)

/* Status REGister */
#define SREG (*(volatile unsigned char *)0x5F)

/* Watchdog Timer Control Register */
#define WDTCSR (*(volatile unsigned char *)0x60)
#define WDTCR (*(volatile unsigned char *)0x60)
#define  WDIF     7
#define  WDIE     6
#define  WDP3     5
```

```
#define  WDCE     4
#define  WDE      3
#define  WDP2     2
#define  WDP1     1
#define  WDP0     0

/* clock prescaler control register */
#define CLKPR (*(volatile unsigned char *)0x61)
#define  CLKPCE   7
#define  CLKPS3   3
#define  CLKPS2   2
#define  CLKPS1   1
#define  CLKPS0   0

/* PRR */
#define PRR  (*(volatile unsigned char *)0x64)
#define  PRTWI    7
#define  PRTIM2   6
#define  PRTIM0   5
#define  PRUSART1 4
#define  PRTIM1   3
#define  PRSPI    2
#define  PRUSART0 1
#define  PRADC    0

/* Oscillator Calibration Register */
#define OSCCAL (*(volatile unsigned char *)0x66)

/* ADC */
#define ADC  (*(volatile unsigned int *)0x78)
#define ADCL (*(volatile unsigned char *)0x78)
#define ADCH (*(volatile unsigned char *)0x79)
#define ADCSRA (*(volatile unsigned char *)0x7A)
#define  ADEN     7
#define  ADSC     6
#define  ADATE    5
#define  ADIF     4
#define  ADIE     3
#define  ADPS2    2
```

```
#define  ADPS1    1
#define  ADPS0    0
#define ADCSRB (*(volatile unsigned char *)0x7B)
#define  ACME     6
#define  ADTS2    2
#define  ADTS1    1
#define  ADTS0    0
#define ADMUX (*(volatile unsigned char *)0x7C)
#define  REFS1    7
#define  REFS0    6
#define  ADLAR    5
#define  MUX4     4
#define  MUX3     3
#define  MUX2     2
#define  MUX1     1
#define  MUX0     0

/* DIDR */
#define DIDR0 (*(volatile unsigned char *)0x7E)
#define  ADC7D    7
#define  ADC6D    6
#define  ADC5D    5
#define  ADC4D    4
#define  ADC3D    3
#define  ADC2D    2
#define  ADC1D    1
#define  ADC0D    0
#define DIDR1 (*(volatile unsigned char *)0x7F)
#define  AIN1D    1
#define  AIN0D    0

/* Timer/Counter1 */
#define ICR1 (*(volatile unsigned int *)0x86)
#define ICR1L (*(volatile unsigned char *)0x86)
#define ICR1H (*(volatile unsigned char *)0x87)
#define OCR1B (*(volatile unsigned int *)0x8A)
#define OCR1BL (*(volatile unsigned char *)0x8A)
#define OCR1BH (*(volatile unsigned char *)0x8B)
#define OCR1A (*(volatile unsigned int *)0x88)
```

```
#define OCR1AL (*(volatile unsigned char *)0x88)
#define OCR1AH (*(volatile unsigned char *)0x89)
#define TCNT1 (*(volatile unsigned int *)0x84)
#define TCNT1L (*(volatile unsigned char *)0x84)
#define TCNT1H (*(volatile unsigned char *)0x85)
#define TCCR1C (*(volatile unsigned char *)0x82)
#define  FOC1A    7
#define  FOC1B    6
#define TCCR1B (*(volatile unsigned char *)0x81)
#define  ICNC1    7
#define  ICES1    6
#define  WGM13    4
#define  WGM12    3
#define  CS12     2
#define  CS11     1
#define  CS10     0
#define TCCR1A (*(volatile unsigned char *)0x80)
#define  COM1A1   7
#define  COM1A0   6
#define  COM1B1   5
#define  COM1B0   4
#define  WGM11    1
#define  WGM10    0

/* Timer/Counter2 */
#define ASSR (*(volatile unsigned char *)0xB6)
#define  EXCLK    6
#define  AS2      5
#define  TCN2UB   4
#define  OCR2AUB  3
#define  OCR2BUB  2
#define  TCR2AUB  1
#define  TCR2BUB  0
#define OCR2B (*(volatile unsigned char *)0xB4)
#define OCR2A (*(volatile unsigned char *)0xB3)
#define TCNT2 (*(volatile unsigned char *)0xB2)
#define TCCR2B (*(volatile unsigned char *)0xB1)
#define  FOC2A    7
#define  FOC2B    6
```

```
#define  WGM22    3
#define  CS22     2
#define  CS21     1
#define  CS20     0
#define TCCR2A (*(volatile unsigned char *)0xB0)
#define  COM2A1   7
#define  COM2A0   6
#define  COM2B1   5
#define  COM2B0   4
#define  WGM21    1
#define  WGM20    0

/* 2-wire SI */
#define TWBR (*(volatile unsigned char *)0xB8)
#define TWSR (*(volatile unsigned char *)0xB9)
#define  TWPS1    1
#define  TWPS0    0
#define TWAR (*(volatile unsigned char *)0xBA)
#define  TWGCE    0
#define TWDR (*(volatile unsigned char *)0xBB)
#define TWCR (*(volatile unsigned char *)0xBC)
#define  TWINT    7
#define  TWEA     6
#define  TWSTA    5
#define  TWSTO    4
#define  TWWC     3
#define  TWEN     2
#define  TWIE     0
#define TWAMR (*(volatile unsigned char *)0xBD)

/* USART0 */
#define UBRR0H (*(volatile unsigned char *)0xC5)
#define UBRR0L (*(volatile unsigned char *)0xC4)
#define UBRR0 (*(volatile unsigned int *)0xC4)
#define UCSR0C (*(volatile unsigned char *)0xC2)
#define  UMSEL01  7
#define  UMSEL00  6
#define  UPM01    5
#define  UPM00    4
```

```
#define  USBS0    3
#define  UCSZ01   2
#define  UCSZ00   1
#define  UCPOL0   0
/* in SPI mode */
#define  UDORD0   2
#define  UCPHA0   1
#define UCSR0B (*(volatile unsigned char *)0xC1)
#define  RXCIE0   7
#define  TXCIE0   6
#define  UDRIE0   5
#define  RXEN0    4
#define  TXEN0    3
#define  UCSZ02   2
#define  RXB80    1
#define  TXB80    0
#define UCSR0A (*(volatile unsigned char *)0xC0)
#define  RXC0     7
#define  TXC0     6
#define  UDRE0    5
#define  FE0      4
#define  DOR0     3
#define  UPE0     2
#define  U2X0     1
#define  MPCM0    0
#define UDR0 (*(volatile unsigned char *)0xC6)

/* USART1 */
#define UBRR1H (*(volatile unsigned char *)0xCD)
#define UBRR1L (*(volatile unsigned char *)0xCC)
#define UBRR1 (*(volatile unsigned int *)0xCC)
#define UCSR1C (*(volatile unsigned char *)0xCA)
#define  UMSEL11  7
#define  UMSEL10  6
#define  UPM11    5
#define  UPM10    4
#define  USBS1    3
#define  UCSZ11   2
#define  UCSZ10   1
```

```
#define  UCPOL1   0
/* in SPI mode */
#define  UDORD1   2
#define  UCPHA1   1
#define UCSR1B (*(volatile unsigned char *)0xC9)
#define  RXCIE1   7
#define  TXCIE1   6
#define  UDRIE1   5
#define  RXEN1    4
#define  TXEN1    3
#define  UCSZ12   2
#define  RXB81    1
#define  TXB81    0
#define UCSR1A (*(volatile unsigned char *)0xC8)
#define  RXC1     7
#define  TXC1     6
#define  UDRE1    5
#define  FE1      4
#define  DOR1     3
#define  UPE1     2
#define  U2X1     1
#define  MPCM1    0
#define UDR1 (*(volatile unsigned char *)0xCE)


/* bits */

/* Port A */
#define  PORTA7   7
#define  PORTA6   6
#define  PORTA5   5
#define  PORTA4   4
#define  PORTA3   3
#define  PORTA2   2
#define  PORTA1   1
#define  PORTA0   0
#define  PA7      7
#define  PA6      6
#define  PA5      5
```

```
#define  PA4      4
#define  PA3      3
#define  PA2      2
#define  PA1      1
#define  PA0      0
#define  DDA7     7
#define  DDA6     6
#define  DDA5     5
#define  DDA4     4
#define  DDA3     3
#define  DDA2     2
#define  DDA1     1
#define  DDA0     0
#define  PINA7    7
#define  PINA6    6
#define  PINA5    5
#define  PINA4    4
#define  PINA3    3
#define  PINA2    2
#define  PINA1    1
#define  PINA0    0

/* Port B */
#define  PORTB7   7
#define  PORTB6   6
#define  PORTB5   5
#define  PORTB4   4
#define  PORTB3   3
#define  PORTB2   2
#define  PORTB1   1
#define  PORTB0   0
#define  PB7      7
#define  PB6      6
#define  PB5      5
#define  PB4      4
#define  PB3      3
#define  PB2      2
#define  PB1      1
#define  PB0      0
```

```
#define  DDB7     7
#define  DDB6     6
#define  DDB5     5
#define  DDB4     4
#define  DDB3     3
#define  DDB2     2
#define  DDB1     1
#define  DDB0     0
#define  PINB7    7
#define  PINB6    6
#define  PINB5    5
#define  PINB4    4
#define  PINB3    3
#define  PINB2    2
#define  PINB1    1
#define  PINB0    0

/* Port C */
#define  PORTC7   7
#define  PORTC6   6
#define  PORTC5   5
#define  PORTC4   4
#define  PORTC3   3
#define  PORTC2   2
#define  PORTC1   1
#define  PORTC0   0
#define  PC7      7
#define  PC6      6
#define  PC5      5
#define  PC4      4
#define  PC3      3
#define  PC2      2
#define  PC1      1
#define  PC0      0
#define  DDC7     7
#define  DDC6     6
#define  DDC5     5
#define  DDC4     4
#define  DDC3     3
```

```
#define  DDC2     2
#define  DDC1     1
#define  DDC0     0
#define  PINC7    7
#define  PINC6    6
#define  PINC5    5
#define  PINC4    4
#define  PINC3    3
#define  PINC2    2
#define  PINC1    1
#define  PINC0    0

/* Port D */
#define  PORTD7   7
#define  PORTD6   6
#define  PORTD5   5
#define  PORTD4   4
#define  PORTD3   3
#define  PORTD2   2
#define  PORTD1   1
#define  PORTD0   0
#define  PD7      7
#define  PD6      6
#define  PD5      5
#define  PD4      4
#define  PD3      3
#define  PD2      2
#define  PD1      1
#define  PD0      0
#define  DDD7     7
#define  DDD6     6
#define  DDD5     5
#define  DDD4     4
#define  DDD3     3
#define  DDD2     2
#define  DDD1     1
#define  DDD0     0
#define  PIND7    7
#define  PIND6    6
```

```
#define  PIND5    5
#define  PIND4    4
#define  PIND3    3
#define  PIND2    2
#define  PIND1    1
#define  PIND0    0

/* PCMSK3 */
#define  PCINT31  7
#define  PCINT30  6
#define  PCINT29  5
#define  PCINT28  4
#define  PCINT27  3
#define  PCINT26  2
#define  PCINT25  1
#define  PCINT24  0
/* PCMSK2 */
#define  PCINT23  7
#define  PCINT22  6
#define  PCINT21  5
#define  PCINT20  4
#define  PCINT19  3
#define  PCINT18  2
#define  PCINT17  1
#define  PCINT16  0
/* PCMSK1 */
#define  PCINT15  7
#define  PCINT14  6
#define  PCINT13  5
#define  PCINT12  4
#define  PCINT11  3
#define  PCINT10  2
#define  PCINT9   1
#define  PCINT8   0
/* PCMSK0 */
#define  PCINT7   7
#define  PCINT6   6
#define  PCINT5   5
#define  PCINT4   4
```

```
#define  PCINT3   3
#define  PCINT2   2
#define  PCINT1   1
#define  PCINT0   0


/* Lock and Fuse Bits with LPM/SPM instructions */

/* lock bits */
#define  BLB12    5
#define  BLB11    4
#define  BLB02    3
#define  BLB01    2
#define  LB2      1
#define  LB1      0

/* fuses low bits */
#define  CKDIV8   7
#define  CKOUT    6
#define  SUT1     5
#define  SUT0     4
#define  CKSEL3   3
#define  CKSEL2   2
#define  CKSEL1   1
#define  CKSEL0   0

/* fuses high bits */
#define  OCDEN    7
#define  JTAGEN   6
#define  SPIEN    5
#define  WDTON    4
#define  EESAVE   3
#define  BOOTSZ1  2
#define  BOOTSZ0  1
#define  BOOTRST  0

/* extended fuses */
#define  BODLEVEL2 2
#define  BODLEVEL1 1
```

```
#define  BODLEVEL0 0


/* Interrupt Vector Numbers */

#define iv_RESET        1
#define iv_INT0         2
#define iv_EXT_INT0     2
#define iv_INT1         3
#define iv_EXT_INT1     3
#define iv_INT2         4
#define iv_EXT_INT2     4
#define iv_PCINT0       5
#define iv_PCINT1       6
#define iv_PCINT2       7
#define iv_PCINT3       8
#define iv_WDT          9
#define iv_TIMER2_COMPA 10
#define iv_TIMER2_COMPB 11
#define iv_TIMER2_OVF   12
#define iv_TIM2_COMPA   10
#define iv_TIM2_COMPB   11
#define iv_TIM2_OVF     12
#define iv_TIMER1_CAPT  13
#define iv_TIMER1_COMPA14
#define iv_TIMER1_COMPB 15
#define iv_TIMER1_OVF   16
#define iv_TIM1_CAPT    13
#define iv_TIM1_COMPA   14
#define iv_TIM1_COMPB   15
#define iv_TIM1_OVF     16
#define iv_TIMER0_COMPA 17
#define iv_TIMER0_COMPB 18
#define iv_TIMER0_OVF   19
#define iv_TIM0_COMPA   17
#define iv_TIM0_COMPB   18
#define iv_TIM0_OVF     19
#define iv_SPI_STC      20
#define iv_USART0_RX    21
```

```
#define iv_USART0_RXC   21
#define iv_USART0_DRE   22
#define iv_USART0_UDRE  22
#define iv_USART0_TX    23
#define iv_USART0_TXC   23
#define iv_ANA_COMP     24
#define iv_ANALOG_COMP  24
#define iv_ADC          25
#define iv_EE_RDY       26
#define iv_EE_READY     26
#define iv_TWI          27
#define iv_TWSI         27
#define iv_SPM_RDY      28
#define iv_SPM_READY    28
#define iv_USART1_RX    29
#define iv_USART1_RXC   29
#define iv_USART1_DRE   30
#define iv_USART1_UDRE  30
#define iv_USART1_TX    31
#define iv_USART1_TXC   31

/*ja*/

#endif
```

Authors' Biographies

STEVEN F. BARRETT

Steven F. Barrett, Ph.D., P.E., received a B.S. in Electronic Engineering Technology from the University of Nebraska at Omaha in 1979, an M.E.E.E. from the University of Idaho at Moscow in 1986, and a Ph.D. from The University of Texas at Austin in 1993. He was formally an active duty faculty member at the United States Air Force Academy, Colorado and is now the Associate Dean of Academic Programs at the University of Wyoming. He is a member of IEEE (senior) and Tau Beta Pi (chief faculty advisor). His research interests include digital and analog image processing, computer–assisted laser surgery, and embedded controller systems. He is a registered Professional Engineer in Wyoming and Colorado. He co-wrote with Dr. Daniel Pack several textbooks on microcontrollers and embedded systems. In 2004, Barrett was named "Wyoming Professor of the Year" by the Carnegie Foundation for the Advancement of Teaching and in 2008 was the recipient of the National Society of Professional Engineers (NSPE) in Higher Education, Engineering Education Excellence Award.

DANIEL J. PACK

Daniel J. Pack, Ph.D., P.E., is the Dean of the College of Engineering and Computer Science at the University of Tennessee, Chattanooga (UTC). Prior to joining UTC, he was Professor and Mary Lou Clarke Endowed Department Chair of the Electrical and Computer Engineering Department at the University of Texas, San Antonio (UTSA). Before his service at UTSA, Dr. Pack was Professor (now Professor Emeritus) of Electrical and Computer Engineering at the United States Air Force Academy (USAFA), CO, where he served as founding Director of the Academy Center for Unmanned Aircraft Systems Research. He received a B.S. in Electrical Engineering, an M.S. in Engineering Sciences, and a Ph.D. in Electrical Engineering from Arizona State University, Harvard University, and Purdue University, respectively. He was a visiting scholar at the Massachusetts Institute of Technology-Lincoln Laboratory. Dr. Pack has co-authored seven textbooks on embedded systems (including *68HC12 Microcontroller: Theory and Applications* and *Embedded Systems: Design and Applications with the 68HC12 and HCS12*) and published over 160 book chapters, technical journal/transactions, and conference papers on unmanned systems, cooperative control, robotics, pattern recognition, and engineering education. He is the recipient of a number of teaching and research awards including Carnegie U.S. Professor of the Year Award, Frank J. Seiler Research Excellence Award, Tau Beta Pi Outstanding Professor Award, Academy Educator Award, and Magoon Award. He is a member of Eta

Kappa Nu (Electrical Engineering Honorary), Tau Beta Pi (Engineering Honorary), IEEE, and the American Society of Engineering Education. He is a registered Professional Engineer in Colorado, serves as Associate Editor of *IEEE Systems Journal*, and is a member on a number of executive advisory or editorial boards including the *Journal of Intelligent & Robotic Systems*, *International Journal of Advanced Robotic Systems*, and *SimCenter Enterprise*. His research interests include unmanned aerial vehicles, intelligent control, automatic target recognition, robotics, and engineering education. E-mail: daniel-pack@utc.edu

Index