MORGAN & CLAYPOOL PUBLISHERS

Arduino II

Systems

Steven F. Barrett

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS

Mitchell A. Thornton, *Series Editor*

Arduino II

Systems

Synthesis Lectures on Digital Circuits and Systems

Editor
Mitchell A. Thornton, ***Southern Methodist University***

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.

Arduino II: Systems
Steven F. Barrett
2020

Arduino I: Getting Started
Steven F. Barrett
2020

Index Generation Functions
Tsutomu Sasao
2019

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

Microcontroller Programming and Interfacing with Texas Instruments MSP430FR2433 and MSP430FR5994 – Part II, Second Edition
Steven F. Barrett and Daniel J. Pack
2019

Microcontroller Programming and Interfacing with Texas Instruments MSP430FR2433 and MSP430FR5994 – Part I, Second 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

Arduino II: Systems

Steven F. Barrett

www.morganclaypool.com

ISBN: 9781681738970 paperback
ISBN: 9781681738987 ebook
ISBN: 9781681738994 hardcover

DOI 10.2200/S01024ED1V01Y202006DCS059

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

Lecture #59
Series Editor: Mitchell A. Thornton, *Southern Methodist University*
Series ISSN
Print 1932-3166 Electronic 1932-3174

Arduino II

Systems

Steven F. Barrett
University of Wyoming, Laramie, WY

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS #59

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

This book is about the Arduino microcontroller and the Arduino concept. The visionary Arduino team of Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, and David Mellis launched a new innovation in microcontroller hardware in 2005, the concept of open-source hardware. Their approach was to openly share details of microcontroller-based hardware design platforms to stimulate the sharing of ideas and promote innovation. This concept has been popular in the software world for many years. In June 2019, Joel Claypool and I met to plan the fourth edition of *Arduino Microcontroller Processing for Everyone!* Our goal has been to provide an accessible book on the rapidly evolving world of Arduino for a wide variety of audiences including students of the ?ne arts, middle and senior high school students, engineering design students, and practicing scientists and engineers. To make the book even more accessible to better serve our readers, we decided to change our approach and provide a series of smaller volumes. Each volume is written to a speci?c audience. This book, *Arduino II: Systems*, is a detailed treatment of the ATmega328 processor and an introduction to C programming and microcontroller-based systems design. *Arduino I: Getting Started* provides an introduction to the Arduino concept. *Arduino III: the Internet of Things* explores Arduino applications in the Internet of Things (IoT).

KEYWORDS

Arduino microcontroller, Arduino UNO R3, microchip AVR ATmega328, programming in C, microcontroller system design

To Mom and Dad.

– Steven

Contents

Preface

This book is about the Arduino microcontroller and the Arduino concept. The visionary Arduino team of Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, and David Mellis launched a new innovation in microcontroller hardware in 2005; the concept of open-source hardware. Their approach was to openly share details of microcontroller–based hardware design platforms to stimulate the sharing of ideas and promote innovation. This concept has been popular in the software world for many years. In June 2019, Joel Claypool and I met to plan the fourth edition of *Arduino Microcontroller Processing for Everyone!* Our goal has been to provide an accessible book on the rapidly changing world of Arduino for a wide variety of audiences including students of the fine arts, middle and senior high school students, engineering design students, and practicing scientists and engineers. To make even the book more accessible to better serve our readers, we decided to change our approach and provide a series of smaller volumes. Each volume is written to a specific audience. This book, *Arduino II: System*, is a detailed treatment of the ATmega328 processor and an introduction to C programming and microcontroller-based systems design. *Arduino I: Getting Started* provides an introduction to the Arduino concept. *Arduino III: the Internet of Things* explores Arduino applications in the Internet of Things (IoT).

APPROACH OF THE BOOK

Arduino II: Systems builds upon the foundation of *Arduino I: Getting Started.* The reader should have a solid grounding in the Arduino UNO R3, the Arduino Development Environment, interfacing techniques to external devices, and writing Arduino sketches. Chapter 1 provides a brief review of some of these concepts and introduces the Microchip ATmega328. This is the processor hosted onboard the Arduino UNO R3. Chapter 2 provides an introduction to programming in C for the novice programmer. It also serves as a good review for the seasoned developer.

Chapters 3–6 provides a "deep dive" into the features of the ATmega328 microcontroller including: analog-to-digital conversion, timing features, serial communications, and interrupts. Each chapter provides the theory of operation, register descriptions, and detailed code examples.

Chapter 7 provides an introduction to embedded system design. It provides a systematic, step-by-step approach on how to design complex systems in a stress free manner. It concludes with several detailed examples.

Steven F. Barrett
August 2020

Acknowledgments

A number of people have made this book possible. I would like to thank Massimo Banzi of the Arduino design team for his support and encouragement in writing the first edition of this book: *Arduino Microcontroller: Processing for Everyone!* In 2005, Joel Claypool of Morgan & Claypool Publishers, invited Daniel Pack and I 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 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.

I would also like to thank Sparkfun, Adafruit, and Microchip for their permission to use images of their products and copyrighted material throughout the text series. 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.

- *Arduino II: Systems* is an independent publication and is not affiliated with, nor has it been authorized, sponsored, or otherwise approved by Microchip.

I would like to dedicate this book to my close friend Dr. Daniel Pack, Ph.D., P.E. In 2000, Daniel suggested that we might write a book together on microcontrollers. I had always wanted to write a book but I thought that's what other people did. With Daniel's encouragement we wrote that first book (and several more since then). Daniel is a good father, good son, good husband, brilliant engineer, great leader, a work ethic second to none, and a good friend. To you, good friend, I dedicate this book. I know that we will do many more together. It is hard to

believe we have been writing together for 20 years. Finally, I would like to thank my wife and best friend of many years, Cindy.

Steven F. Barrett
August 2020

CHAPTER 1

Getting Started

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

- successfully download and execute a simple program using the Arduino Development Environment;
- describe the key features of the Arduino Development Environment;
- name and describe the different features aboard the Arduino UNO R3 processor board; and
- discuss the features and functions of the Microchip ATmega328.

1.1 OVERVIEW

Welcome to the world of Arduino! The Arduino concept of open-source hardware was developed by the visionary Arduino team of Massimo Banzi, David Cuartilles, Tom Igoe, Gianluca Martino, and David Mellis in Ivrea, Italy. The team's goal was to develop a line of easy-to-use microcontroller hardware and software such that processing power would be readily available to everyone.

We assume you have a solid grounding in the Arduino UNO R3, the Arduino Development Environment, interfacing techniques, and Arduino sketch writing. The chapter begins with a brief review of some of these concepts.

We use a top-down design approach. We begin with the "big picture" of the chapter. We then discuss the Ardunio Development Environment and how it may be used to quickly develop a program (sketch) for the Arduino UNO R3. We then provide an overview of the hardware features of the Arduino UNO R3 evaluation board which hosts the Microchip ATmega328 processor.

1.2 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. As an alternative, the Arduino Development Environment (ADE) provides a user-friendly interface to quickly develop a program, transform the program to machine code, and then load the machine code into the Arduino processor in several simple steps, as shown in Figure 1.1.

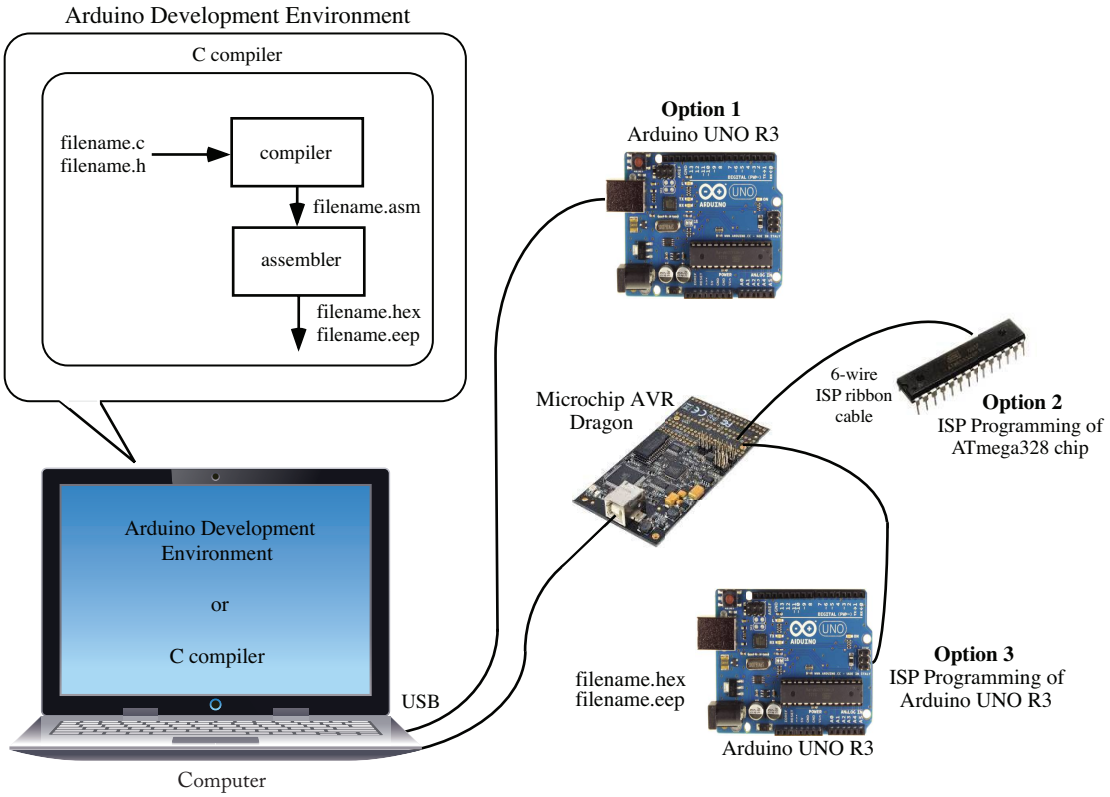


Figure 1.1: Programming the Arduino processor board. (Arduino illustrations used with permission of the Arduino Team (CC BY-NC-SA) [www.arduino.cc]. Microchip AVR Dragon illustration used with permission of Microchip, Inc. [www.microchip.com].)

The first version of the ADE was released in August 2005. It was developed at the Interaction Design Institute in Ivrea, Italy to allow students the ability to quickly put processing power to use in a wide variety of projects. Since that time, updated versions incorporating new features have been released on a regular basis [www.arduino.cc].

At its most fundamental level, the ADE is a user-friendly interface to allow one to quickly write, load, and execute code on a microcontroller. A barebones program need only consist of a setup() and loop() function. The ADE adds the other required pieces such as header files and the main program construct. The ADE is written in Java and has its origins in the Processor programming language and the Wiring Project [www.arduino.cc].

The ADE is hosted on a laptop or personal computer (PC). Once the Arduino program, referred to as a sketch, is written; it is verified and uploaded to the Arduino UNO R3 evaluation board. Alternatively, a program may be written in C using a compiler. The compiled code can

be uploaded to the Arduino UNO R3 using a programming pod such as the Microchip AVR Dragon.

1.3 ARDUINO QUICKSTART

To get started using an Arduino-based platform, you will need the following hardware and software:

- an Arduino-based hardware processing platform;
- the appropriate interface cable from the host PC or laptop to the Arduino platform;
- an Arduino compatible power supply; and
- the Arduino software.

Interface cable. The Arduino UNO R3 connects to the host laptop or PC via a USB cable (Type A male to Type B female). **Power supply.** The Arduino processing boards may be powered from the USB port during project development. However, it is highly recommended that an external power supply be employed. This will allow developing projects beyond the limited electrical current capability of the USB port. For the UNO R3 platform, Arduino (www.arduino.cc) recommends a power supply from 7–12 VDC with a 2.1-mm center positive plug. A power supply of this type is readily available from a number of electronic parts supply companies. For example, the Jameco #133891 power supply is a 9 VDC model rated at 300 mA and equipped with a 2.1-mm center positive plug. It is available for under US$10. The UNO has an onboard voltage regulators that maintain the incoming power supply voltage to a stable 5 VDC.

1.3.1 QUICK START GUIDE

The ADE may be downloaded from the Arduino website's front page at www.arduino.cc. Versions are available for Windows, Mac OS X, and Linux. Provided below is a quick start step-by-step approach to blink an onboard LED.

- Download the ADE from www.arduino.cc.
- Connect the Arduino UNO R3 processing board to the host computer via a USB cable (A male to B male).
- Start the ADE.
- Under the Tools tab select the evaluation **Board** you are using and the **Port** that it is connected to.
- Type the following program.

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

#define LED_PIN 13

void setup()
{
pinMode(LED_PIN, OUTPUT);
}

void loop()
{
digitalWrite(LED_PIN, HIGH);
delay(500);                    //delay specified in ms
digitalWrite(LED_PIN, LOW);
delay(500);
}

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

- Upload and execute the program by asserting the "Upload" (right arrow) button.
- The onboard LED should blink at 1-s intervals.

With the ADE downloaded and exercised, let's take a closer look at its features.

1.3.2 ARDUINO DEVELOPMENT ENVIRONMENT OVERVIEW

The ADE is illustrated in Figure 1.2. The ADE contains a text editor, a message area for displaying status, a text console, a tool bar of common functions, and an extensive menuing system. The ADE also provides a user-friendly interface to the Arduino processor board which allows for a quick upload of code. This is possible because the Arduino processing boards are equipped with a bootloader program.

A close up of the Arduino toolbar is provided in Figure 1.3. The toolbar provides single button access to the more commonly used menu features. Most of the features are self-explanatory. As described in the previous section, the "Upload" button compiles your code and uploads it to the Arduino processing board. The "Serial Monitor" button opens the serial monitor feature. The serial monitor feature allows text data to be sent to and received from the Arduino processing board.

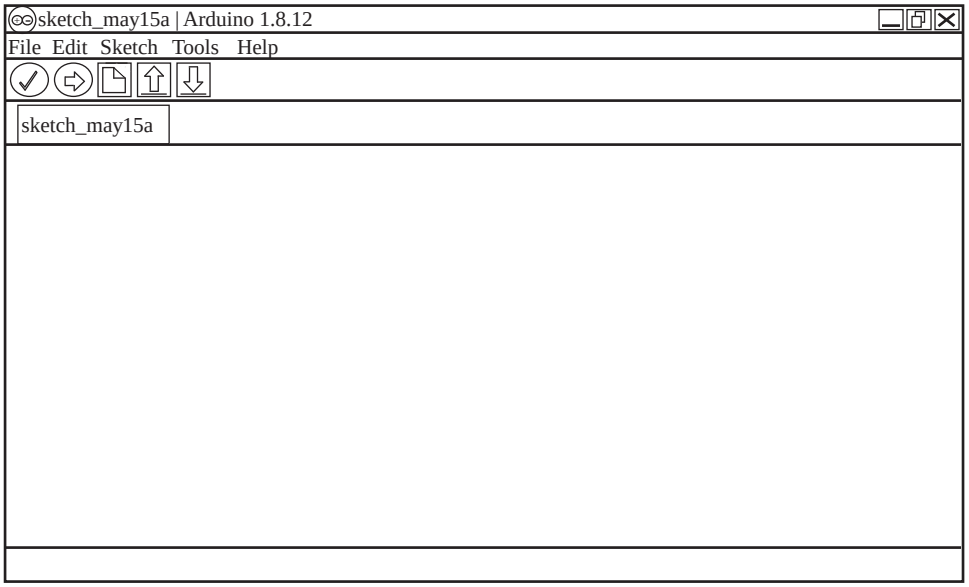


Figure 1.2: Arduino Development Environment [www.arduino.cc].

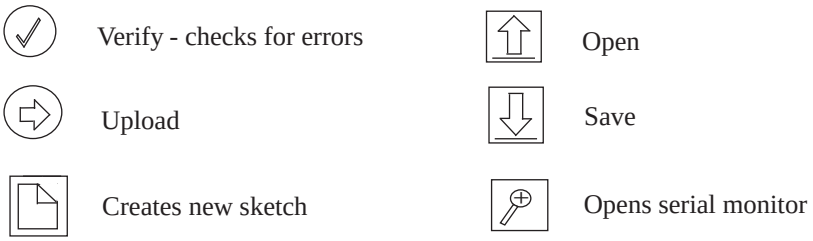


Figure 1.3: Arduino Development Environment buttons.

1.3.3 SKETCHBOOK CONCEPT

In keeping with a hardware and software platform for students of the arts, the Arduino environment employs the concept of a sketchbook. An artist maintains their works in progress in a sketchbook. Similarly, programs are maintained within a sketchbook in the Arduino environment. Furthermore, we refer to individual programs as sketches. An individual sketch within the sketchbook may be accessed via the Sketchbook entry under the file tab.

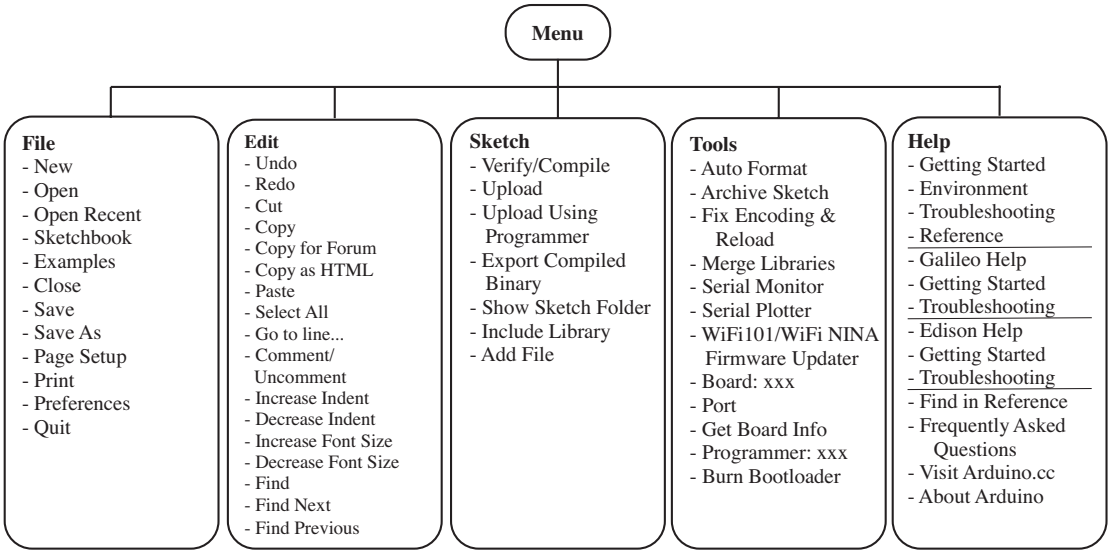


Figure 1.4: Arduino Development Environment menu [www.arduino.cc].

1.3.4 ARDUINO SOFTWARE, LIBRARIES, AND LANGUAGE REFERENCES

The ADE has a number of built-in features. Some of the features may be directly accessed via the ADE drop down toolbar illustrated in Figure 1.2. Provided in Figure 1.4 is a handy reference to show the available features. The toolbar provides a wide variety of features to compose, compile, load, and execute a sketch.

1.3.5 WRITING AN ARDUINO SKETCH

The basic format of the Arduino sketch consists of a "setup" and a "loop" function. The setup function is executed once at the beginning of the program. It is used to configure pins, declare variables and constants, etc. The loop function will execute sequentially step-by-step. When the end of the loop function is reached it will automatically return to the first step of the loop function and execute again. This goes on continuously until the program is stopped.

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

void setup()
  {
  //place setup code here
  }
```

```
void loop()
  {
  //main code steps are provided here
  :
  :

  }

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

Example: Let's revisit the sketch provided earlier in the chapter.

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

#define LED_PIN 13                      //name pin 13 LED_PIN

void setup()
{
pinMode(LED_PIN, OUTPUT);               //set pin to output
}

void loop()
{
digitalWrite(LED_PIN, HIGH);            //write pin to logic high
delay(500);                             //delay specified in ms
digitalWrite(LED_PIN, LOW);             //write to logic low
delay(500);                             //delay specified in ms
}

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

In the first line the #define statement links the designator "LED_PIN" to pin 13 on the Arduino processor board. In the setup function, LED_PIN is designated as an output pin. Recall the setup function is only executed once. The program then enters the loop function that is executed sequentially step-by-step and continuously repeated. In this example, the LED_PIN is first set to logic high to illuminate the LED onboard the Arduino processing board. A 500 ms delay then occurs. The LED_PIN is then set low. A 500 ms delay then occurs. The sequence then repeats.

Even the most complicated sketches follow the basic format of the setup function followed by the loop function. To aid in the development of more complicated sketches, the ADE has many built-in features that may be divided into the areas of structure, variables, and functions.

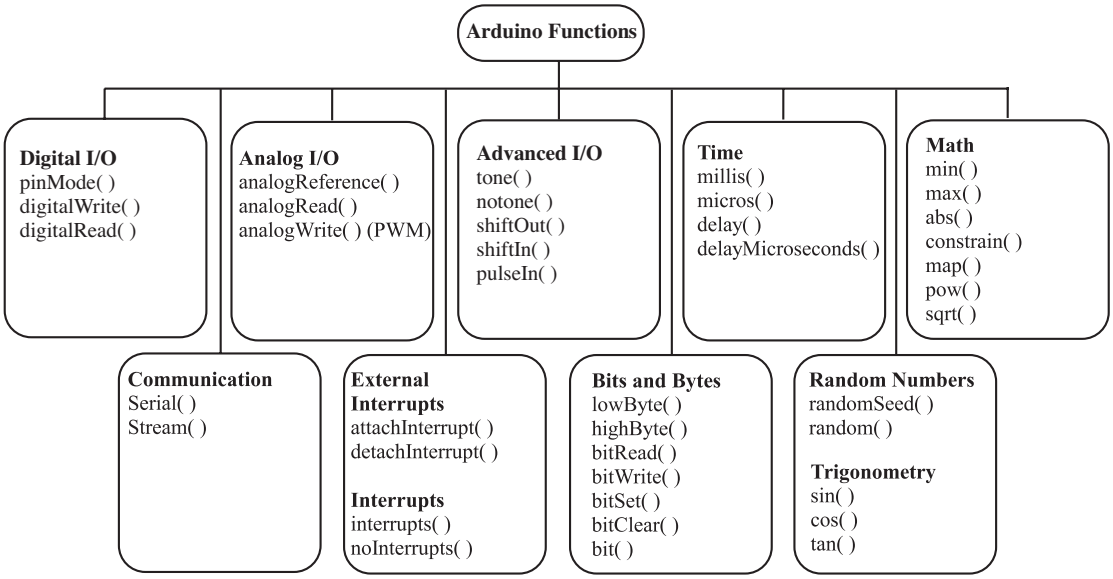


Figure 1.5: Arduino Development Environment functions [www.arduino.cc].

The structure and variable features follow rules similar to the C programming language which is discussed in Chapter 2. The built-in functions consists of a set of pre-defined activities useful to the programmer. These built-in functions are summarized in Figure 1.5.

There are many program examples available to allow the user to quickly construct a sketch. These programs are summarized in Figure 1.6. Complete documentation for these programs is available at the Arduino homepage [www.arduino.cc]. This documentation is easily accessible via the Help tab on the ADE toolbar. This documentation will not be repeated here. With the Arduino open source concept, users throughout the world are constantly adding new built-in features. As new features are added, they are released in future ADE versions. As an Arduino user, you too may add to this collection of useful tools. Throughout the remainder of the book we use both the ADE and also several C compilers to program the Arduino UNO R3. In the next section we get acquainted with the features of the UNO R3.

1.4 ARDUINO UNO R3 PROCESSING BOARD

The Arduino UNO R3 processing board is illustrated in Figure 1.7. Working clockwise from the left, the board is equipped with a USB connector to allow programming the processor from a host personal computer (PC) or laptop. The board may also be programmed using In System Programming (ISP) techniques. A 6-pin ISP programming connector is on the opposite side of the board from the USB connector.

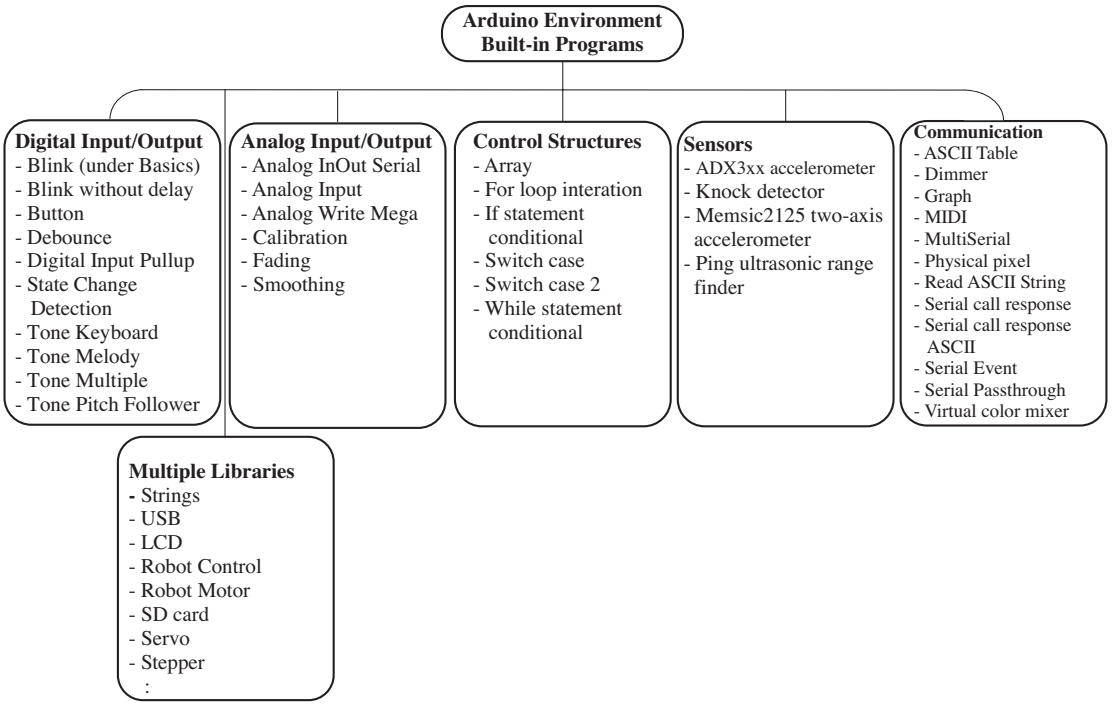


Figure 1.6: Arduino Development Environment built-in features [www.arduino.cc].

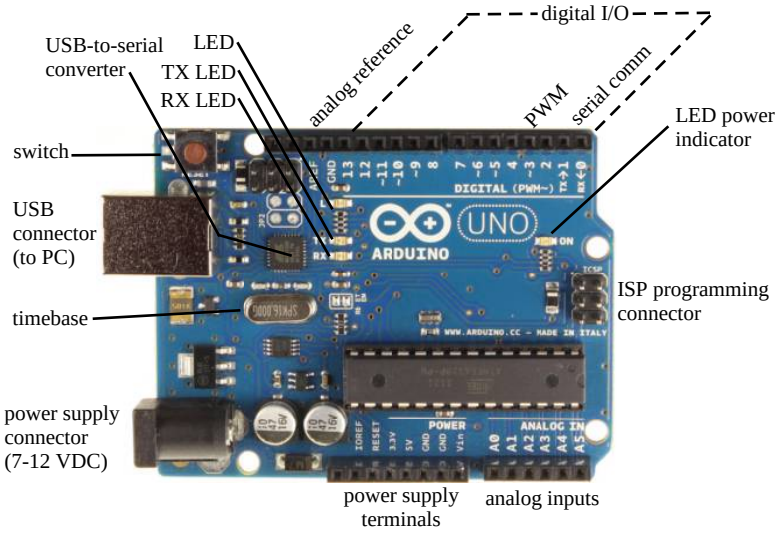


Figure 1.7: Arduino UNO R3 layout. (Figure adapted and used with permission of Arduino Team (CC BY-NC-SA) [www.arduino.cc].)

The board is equipped with a USB-to-serial converter to allow compatibility between the host PC and the serial communications systems aboard the Microchip ATmega328 processor. The UNO R3 is also equipped with several small surface mount light emitting diodes (LEDs) to indicate serial transmission (TX) and reception (RX) and an extra LED for project use. The header strip at the top of the board provides access for an analog reference signal, pulse width modulation (PWM) signals, digital input/output (I/O), and serial communications. The header strip at the bottom of the board provides analog inputs for the analog-to-digital (ADC) system and power supply terminals. Finally, the external power supply connector is provided at the bottom left corner of the board. The top and bottom header strips conveniently mate with an Arduino shield to extend the features of the Arduino host processor.

1.5 ARDUINO UNO R3 OPEN SOURCE SCHEMATIC

The entire line of Arduino products is based on the visionary concept of open-source hardware and software. That is, hardware and software developments are openly shared among users to stimulate new ideas and advance the Arduino concept. In keeping with the Arduino concept, the Arduino team openly shares the schematic of the Arduino UNO R3 processing board; see Figure 1.8.

1.6 ARDUINO UNO R3 HOST PROCESSOR – THE ATMEGA328

The host processor for the Arduino UNO R3 is the Microchip ATmega328. The "328" is a 28 pin, 8-bit microcontroller. The architecture is based on the Reduced Instruction Set Computer (RISC) concept which allows the processor to complete 20 million instructions per second (MIPS) when operating at 20 MHz. The "328" is equipped with a wide variety of features as shown in Figure 1.9. The pin out diagram and block diagram for this processor are provided in Figures 1.10 and 1.11. The features may be conveniently categorized into the following systems:

- memory system,
- port system,
- timer system,
- analog-to-digital converter (ADC),
- interrupt system, and
- serial communications.

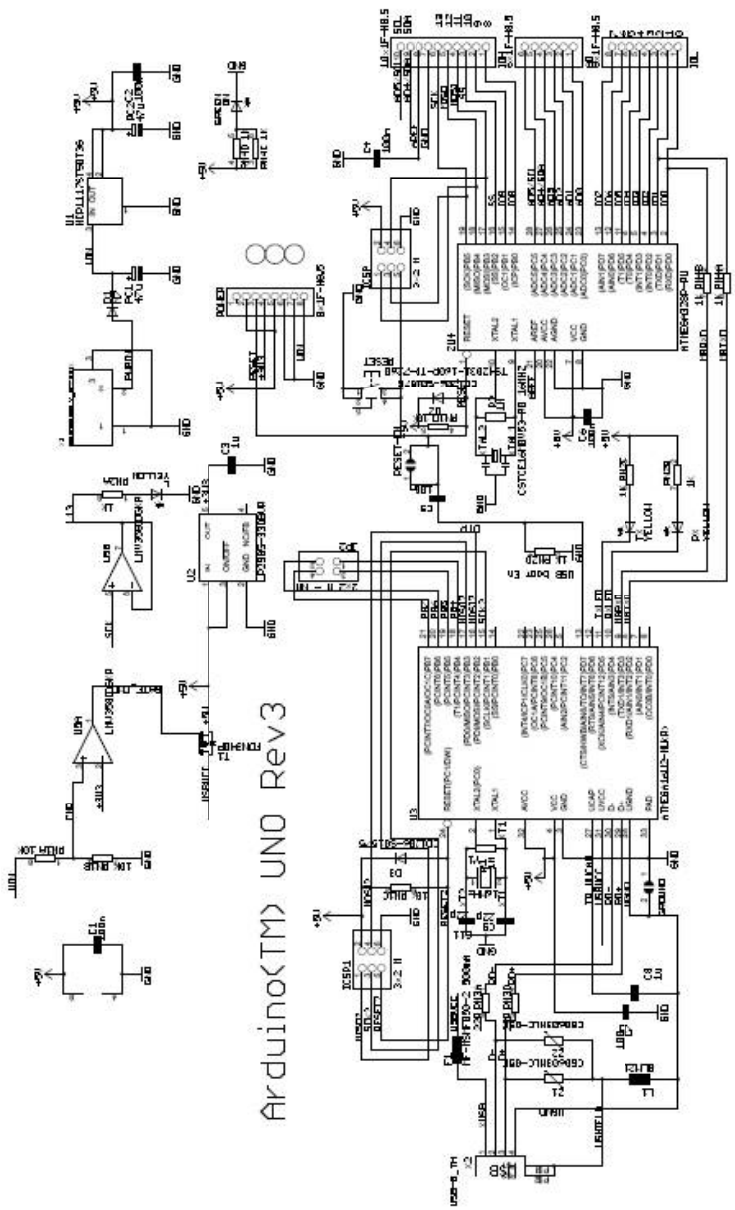


Figure 1.8: Arduino UNO R3 open source schematic. (Figure adapted and used with permission of the Arduino Team (CC BY-NC-SA) [www.arduino.cc].)

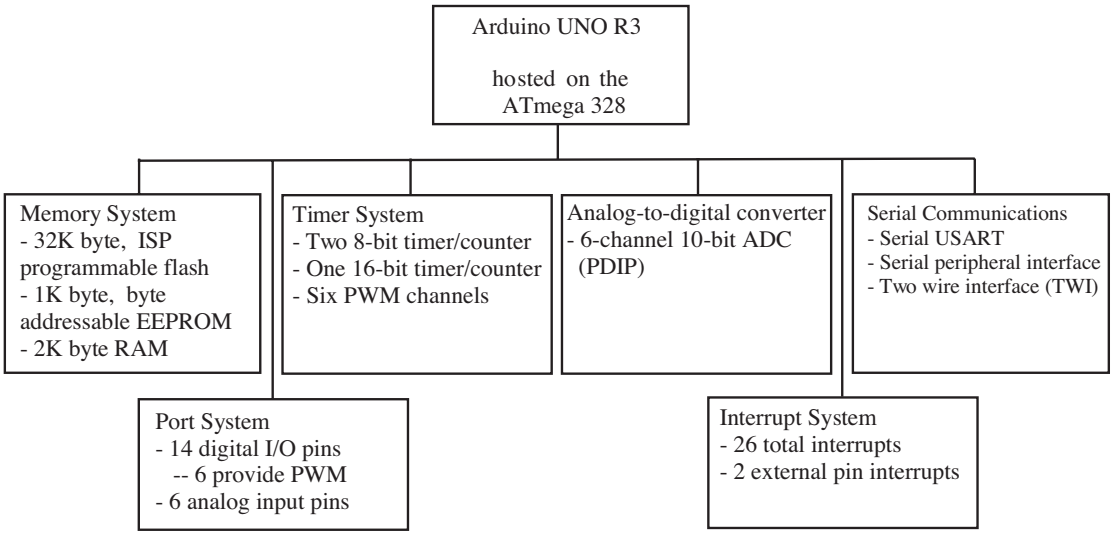


Figure 1.9: Arduino UNO R3 systems.

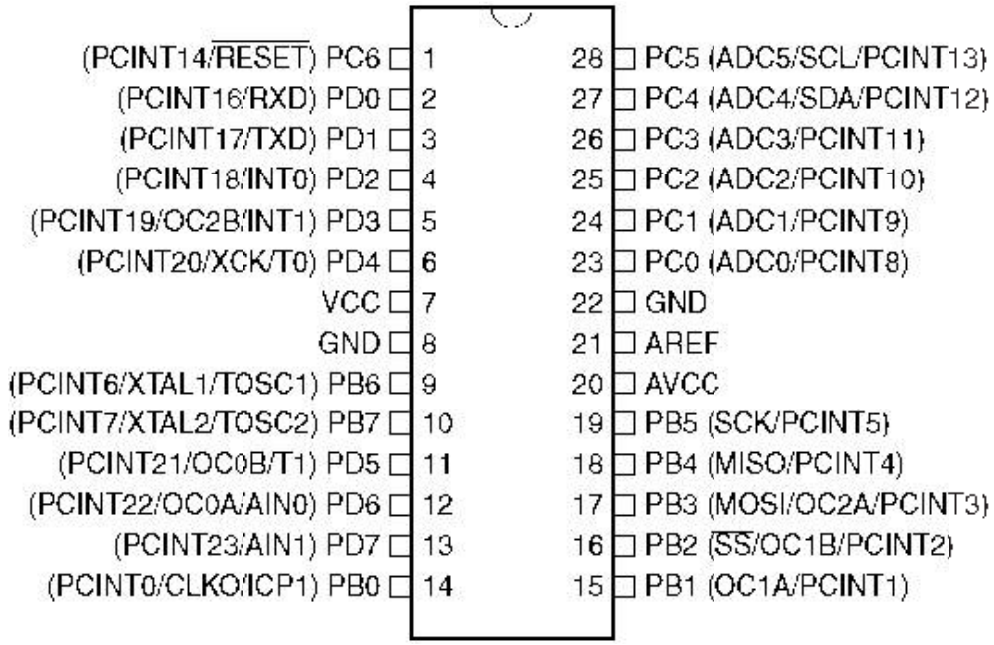


Figure 1.10: ATmega328 pin out. (Figure used with permission of Microchip, Inc. [www.microchip.com].)

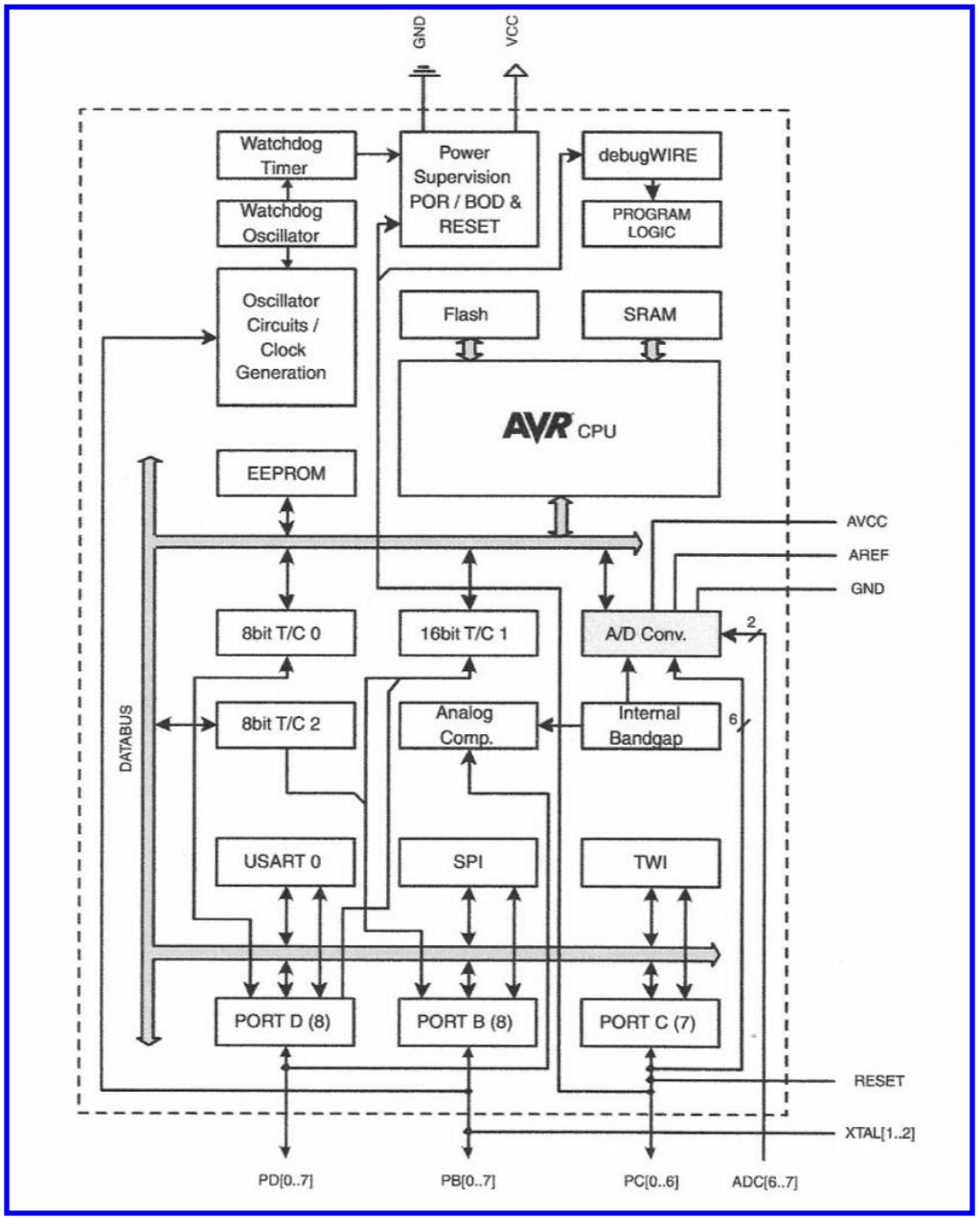


Figure 1.11: ATmega328 block diagram. (Figure used with permission of Microchip, Inc. [www.microchip.com].)

1.6.1 ATMEGA328 MEMORY

The ATmega328 is equipped with three main memory sections: flash electrically erasable programmable read only memory (EEPROM), static random access memory (SRAM), and byte-addressable EEPROM. We discuss each memory component in turn.

1.6.1.1 ATmega328 in – System Programmable Flash EEPROM

Bulk programmable flash EEPROM is 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 even when microcontroller power is lost. The ATmega328 is equipped with 32 K bytes of onboard reprogrammable flash memory. This memory component is organized into 16 K locations with 16 bits at each location.

1.6.1.2 ATmega328 Byte-Addressable EEPROM

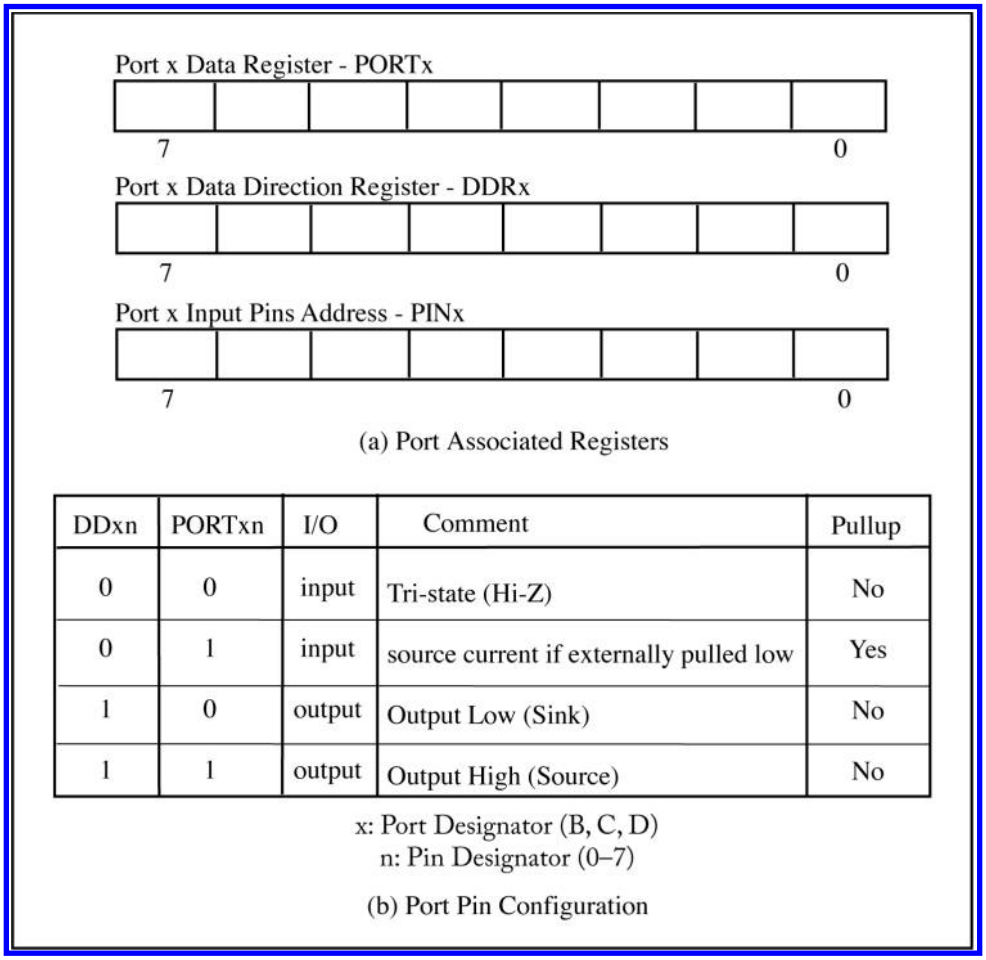
Byte-addressable EEPROM memory is used to permanently store and recall variables during program execution. It too is nonvolatile. It is especially useful for logging system malfunctions and fault data during program execution. It is also useful for storing data that must be retained during a power failure 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 ATmega328 is equipped with 1024 bytes of EEPROM.

1.6.1.3 ATmega328 Static Random Access Memory (SRAM)

Static RAM memory 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 ATmega328 is equipped with 2 K bytes of SRAM. A small portion of the SRAM is set aside for the general-purpose registers used by the processor and also for the input/output and peripheral subsystems aboard the microcontroller. A header file provides the link between register names used in a program and their physical description and location in memory. During program execution, RAM is used to store global variables, support dynamic memory allocation of variables, and to provide a location for the stack.

1.6.2 ATMEGA328 PORT SYSTEM

The Microchip ATmega328 is equipped with three, 8-bit general purpose, digital input/output (I/O) ports designated PORTB (8 bits, PORTB[7:0]), PORTC (7 bits, PORTC[6:0]), and PORTD (8 bits, PORTD[7:0]). As shown in Figure 1.12, each port has three registers associated with it:



(a) Port Associated Registers

DDxn	PORTxn	I/O	Comment	Pullup
0	0	input	Tri-state (Hi-Z)	No
0	1	input	source current if externally pulled low	Yes
1	0	output	Output Low (Sink)	No
1	1	output	Output High (Source)	No

x: Port Designator (B, C, D)
n: Pin Designator (0–7)

(b) Port Pin Configuration

Figure 1.12: ATmega328 port configuration registers [www.microchip.com].

- 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 1.12b 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. If selected for output, the pin may be further configured for either logic low or logic high.

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.

1.6.3 ATMEGA328 INTERNAL SYSTEMS

In this section, we provide a brief overview of the internal features of the ATmega328. It should be emphasized that these features are the internal systems contained within the confines of the microcontroller chip. These built-in features allow complex and sophisticated tasks to be accomplished by the microcontroller.

1.6.3.1 ATmega328 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–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 provide a time base for all peripheral subsystems. The ATmega328 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. You may choose from several different internal fixed clock operating frequencies.

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 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, if the microcontroller will be interfaced to external peripheral devices either a ceramic resonator or a crystal oscillator should be used as a time base. Both are available in a wide variety of operating frequencies. The maximum operating frequency of the ATmega328P is 20 MHz [www.microchip.com].

1.6.3.2 ATmega328 Timing Subsystem

The ATmega328 is equipped with a complement of timers which 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 ATmega328 is equipped with two 8-bit timer/counters and one 16-bit counter.

1.6.3.3 Pulse Width Modulation Channels

A 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 ATmega328 is equipped with four PWM channels. The PWM channels coupled with the flexibility of dividing the time base down to different PWM subsystem clock source frequencies allows 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.

1.6.3.4 ATmega328 Serial Communications

The ATmega328 is equipped with a variety 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 these systems have in common is the serial transmission of data. In a serial communications transmission, serial data is sent a single bit at a time from transmitter to receiver.

ATmega328 Serial USART The serial USART may be used for full duplex (two-way) communication between a receiver and transmitter. This is accomplished by equipping the ATmega328 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 ATmega328 USART is quite flexible. It has the capability to be set to different 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 ATmega328 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.

ATmega328 Serial Peripheral Interface (SPI) The ATmega328 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 to 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 since it is providing the synchronizing clock source between the transmitter and the receiver. The receiver is designated as the slave.

ATmega328 Two-Wire Serial Interface (TWI) The TWI subsystem allows the system designer to network 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 interconnected. 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.

1.6.3.5 ATmega328 Analog to Digital Converter (ADC)

The ATmega328 is equipped with an eight-channel analog to digital converter (ADC) subsystem. The ADC converts an analog signal from the outside world into a binary representation suitable for use by the microcontroller. The ATmega328 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 ATmega328 with a voltage resolution of approximately 4.88 mV.

1.6.3.6 ATmega328 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 both inside and outside the microcontroller. When these higher priority events occur, the microcontroller suspends normal operation and executes event specific actions contained within an interrupt service routine (ISR). Once the higher priority event has been serviced by the ISR, the microcontroller returns and continues processing the normal program.

The ATmega328 is equipped with a complement of 26 interrupt sources. Two of the interrupts are provided for external interrupt sources while the remaining interrupts support the efficient operation of peripheral subsystems aboard the microcontroller.

1.7 SUMMARY

The goal of this chapter was to provide a tutorial on how to begin programming. We used a top-down design approach. We began with the "big picture" of the chapter followed by an overview of the Arduino Development Environment. Throughout the chapter, we provided examples and also provided references to a number of excellent references.

1.8 REFERENCES

[1] Arduino homepage. www.arduino.cc

[2] *Microchip ATmega328 PB AVR Microcontroller with Core Independent Peripherals and Pico Power Technology DS40001906C*. Microchip Technology Incorporation, 2018. www.microchip.com

1.9 CHAPTER PROBLEMS

1. Describe the steps in writing a sketch and executing it on an Arduino UNO R3 processing board.
2. What is the serial monitor feature used for in the Arduino Development Environment?
3. Describe what variables are required and returned and the basic function of the following built-in Arduino functions: Blink, Analog Input.
4. Sketch a block diagram of the ATmega328 and its associated systems. Describe the function of each system.
5. Describe the different types of memory components within the ATmega328. Describe applications for each memory type.
6. Describe the three different register types associated with each port.
7. How may the features of the Arduino UNO R3 be extended?
8. Discuss different options for the ATmega328 time base. What are the advantages and disadvantages of each type? Construct a summary table.
9. Discuss the three types of serial communication systems aboard the ATmega328. Research an application for each system.
10. What is the difference between an ADC and digital-to-analog converter (DAC). Which one is aboard the ATmega328?
11. How does the Arduino UNO R3 receive power? Describe in detail.
12. What is the time base for the Arduino UNO R3? At what frequency does it operate? How many clock pulses per second does the time base provide? What is the time between pulses?
13. What is meant by the term open source?
14. What is the maximum operating frequency of the ATmega328? What is the lowest operating frequency of the ATmega328?
15. What is the range of operating voltages that may be applied to the ATmega328?

CHAPTER 2

Programming

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

- describe the key components of a program;
- specify the size of different variables within the C programming language;
- define the purpose of the main program;
- explain the importance of using functions within a program;
- write functions that pass parameters and return variables;
- describe the function of a header file;
- discuss different programming constructs used for program control and decision processing; and
- write programs in C for use on the Arduino UNO R3 board.

2.1 OVERVIEW

We begin by revisiting the big picture of how to program the Arduino UNO R3 provided in Chapter 1. This will help provide an overview of how chapter concepts fit together. It also introduces terms used in writing, editing, compiling, loading, and executing a program.

As discussed in Chapter 1, most microcontrollers are programmed with some variant of the C programming language.[1] 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 Arduino UNO R3. The job of the compiler is to transform the program written by the program writer (filename.c and filename.h) into machine code (filename.hex) suitable for loading into the processor. The "filename.c" file contains the main program while the "filename.h" file(s) contain functions grouped by type. This technique provides an orderly method of organizing a large program.

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.

[1]This chapter was adapted with permission from *Microcontroller Programming and Interfacing*, S. F. Barrett and D. J. Pack, Morgan & Claypool Publishers, 2011. It has been adapted for use with the ATmega328.

Here the program source files are first 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. The assembler transforms the assembly language source file (filename.asm) to machine code (filename.hex) suitable for loading to the Arduino processor. The machine code file can now be downloaded to the Arduino processor using the ISP features of the Microchip AVR Dragon board. In the examples provided at the end of this chapter we demonstrate how to program the ATmega328 aboard the UNO using two different compilers (the Atmel® Studio [www.microchip.com] and the ImageCraft JumpStartC for AVR [www.ImageCraft.com]) and ISP programming techniques.

In the next section, we describe the parts of a C program.

2.2 ANATOMY OF A C PROGRAM

Programs written in C for a microcontroller have a repeatable format. Slight variations exist but many follow the format provided.

```
//*****************************************************************
//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.

Often function definitions are placed in accompanying header files.

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

Let's take a closer look at each piece.

2.2.1 COMMENTS

Comments are used throughout the program to document what and how things were accomplished within a program. The comments help you reconstruct your work maybe months or even years after completing the program. 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. A multi-line comment can be constructed using a /* at the beginning of the comment and a */ at the end of the comment. The multi-line comment technique may be used to block out portions of code during troubleshooting.

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
- program description.

2.2.2 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 is provided with the compiler and provides the name of each register used within the microcontroller. It also provides the link between a specific register's name within 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 will also be other header files to include in your program such as the "math.h" file when programming with advanced math functions. Also, the function bodies that you write for your program may be placed in other files. This is especially useful in large programs that contain many functions. The functions may be grouped into different files by category. This allows a large program to be subdivided into more manageable, smaller files. These files can be then connected to the main program file with the #include statements.

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

```
//include files
#include<file_name1.h>
#include<file_name2.h>
#include<file_name3.c>
  :
  :
```

2.2.3 FUNCTIONS

In Chapter 7, we discuss the top-down design, bottom-up implementation approach to designing 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.

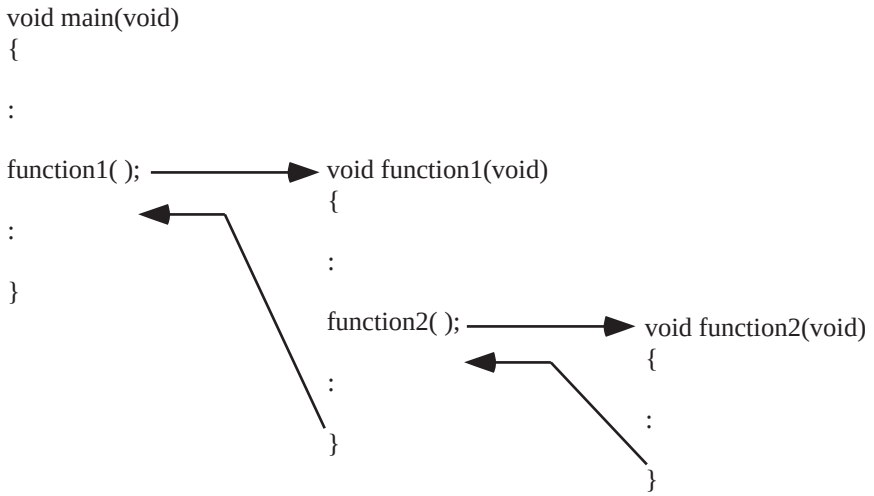


Figure 2.1: Function calling.

We use the same approach when writing computer programs. At the highest level is the main program which calls functions that have a defined action. When a function is called, program control is released from the main program to the function. Once the function is complete, program control reverts to the main program. Functions may in turn call other functions, as shown in Figure 2.1.

Use of the top-down design, bottom-up implementation approach results in a collection of functions that may be reused in various projects. Also, the program is subdivided into doable pieces, each with a defined action. This makes writing the program easier. Also, it is easier to modify a program since every action is in a known location.

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

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

Function prototypes are provided early in the program as previously shown in the program template. The function prototype provides the name of the function and any variables required by the function and any 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 does not send back a variable 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 this 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 variable returned by the function. The last line of the function contains a "return" statement. Here a variable 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, 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 function body 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. Associated with each port is a register called the data direction

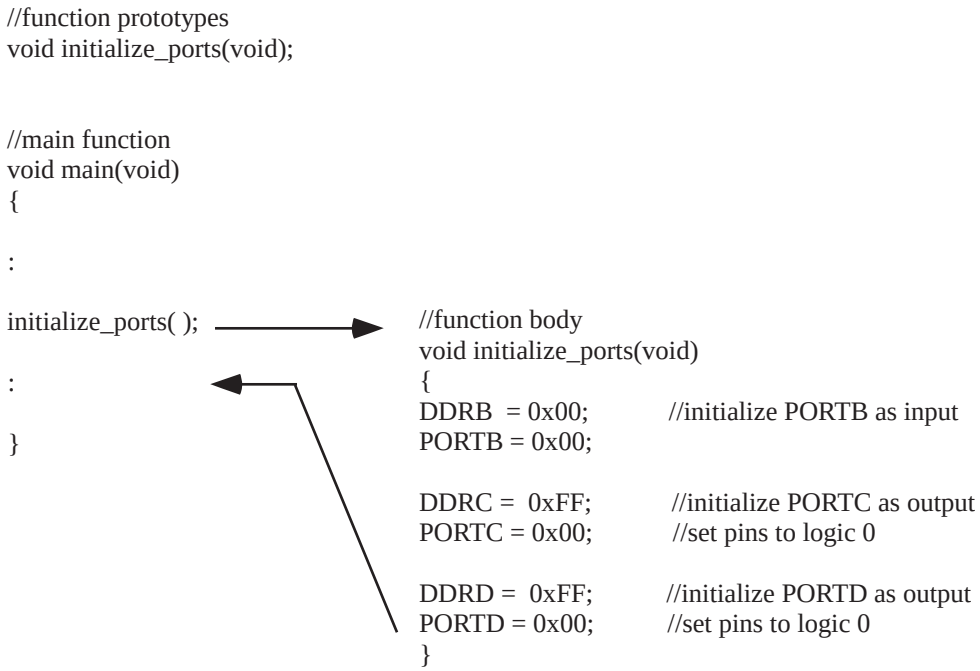


Figure 2.2: Configuring ports.

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.

2.2.4 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:

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

Type	Size	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 (ImageCraft [2]).

2.2.5 INTERRUPT HANDLER DEFINITIONS

Interrupts are functions that are written by the programmer but usually called by a specific hardware event during system operation. We discuss interrupts and how to properly configure them in Chapter 6.

2.2.6 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 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].

A global variable is declared using the following format provided below. The type of the variable is specified, followed by its name, and an initial value if desired. When programming microcontrollers, it is important to know the number of bits used to store the variable and where the 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 insure your assignment statements are balanced for accurate and predictable results. The modifier "unsigned" indicates all bits will be used to specify the magni-

tude of the argument. Signed variables will use the left most bit to indicate the polarity (±) of the argument.

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

2.2.7 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.

2.3 FUNDAMENTAL PROGRAMMING CONCEPTS

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

2.3.1 OPERATORS

There are many operators provided in the C language. An abbreviated list of common operators are provided 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 provided. The precedence column indicates the priority of the operator in a program statement containing multiple operators. Only the fundamental operators are provided. For more information on this topic, see Barrett and Pack in the Reference section at the end of the chapter.

2.3.1.1 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 yield a different result from the earlier expression.

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 [1].)

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 [1].)

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 insure the left- and the right-hand side of the equation have the same type of arguments. If not, unpredictable results may occur.

2.3.1.2 Arithmetic Operations

The arithmetic operations provide for basic math operations using the various variables described in the previous section. As described in the previous section, the assignment operator (=) is used to assign the argument(s) on the right-hand side of an equation to the left-hand side variable.

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

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

sum = variable1 + variable2;
return sum;
}
```

2.3.1.3 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.

2.3.1.4 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 eight-bit unsigned char and assigned the hexadecimal value of $(73)_{16}$. In binary notation this is expressed as $(0111\_0011)_2$. The << 2 operator provides a left shift of the argument by two places. After two left shifts of $(73)_{16}$, the result is $(cc)_{16}$ and will be 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, respectively.

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

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

Figure 2.6: Bit twiddling (ImageCraft [2]).

2.3.1.5 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 common 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 provides bit twiddling examples on how individual bits may be manipulated without affecting other bits using bitwise and unary operators. The information provided here was extracted from the ImageCraft JumpStart C for AVR compiler documentation [www.ImageCraft.com].

2.3.2 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 initialization,
- loop termination testing, and
- the 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)
  {


  }
```

2.3.3 DECISION PROCESSING

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

- the **if** statement,
- the **if-else** construct,
- the **if-else if-else** construct, and the
- **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: A debounced tact switch is attached to PORTC pin 2 (PORTC[2]) of an ATmega328. We want to illuminate an LED connected to PORT B pin 1 (PORTB[1]) the first time the switch is pressed.[2] Provided below is a code snapshot to implement this task.

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

if((PINC & 0x04)== 0x04)      //Test PORTC[2] for logic one
   {
   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 ((PINC & 0x04)== 0x04)     //Test PORTC[2] for logic one
   {
   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 switch system. In this example, three debounced switches are connected to 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 ((PINC & 0x04)== 0x04)     //Test PORTC[2] for logic one
   {
   PORTB = 0x04;              //illuminate LED on PORTB[2]
   }
else if ((PINC & 0x02)== 0x02)//Test PORTC[1] for logic one
```

[2]Recall that pin numbering on ports begin with pin 0 and go through pin 7.

```
   {
   PORTB = 0x02;              //illuminate the LED on PORTB[1]
   }
else if ((PINC & 0x01)== 0x01)//Test PORTC[0] for logic one
   {
   PORTB = 0x01;              //illuminate the LED on PORTB[0]
   }
else
   {
   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. The alternatives are processed in the order specified by the switch statement. If no match is found, the default case is executed.

Example: Suppose eight debounced pushbutton switches are connected to PORTB. 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_PORTB = PINB;                  //get current value of PORTB

if(new_PORTB != old_PORTB)         //check for status change PORTB

switch(new_PORTB)
  {                                //process change in PORTB input
  case 0x01:                       //PB0
               :                   //PB0 related actions
            break;

  case 0x02:                       //PB1
               :                   //PB1 related actions
            break;
```

```
  case 0x04:                          //PB2
              :                       //PB2 related actions
            break;

  case 0x08:                          //PB3
              :                       //PB3 related actions
            break;

  case 0x10:                          //PB4
              :                       //PB4 related actions
            break;

  case 0x20:                          //PB5
              :                       //PB5 related actions
            break;

  case 0x40:                          //PB6
              :                       //PB6 related actions
            break;

  case 0x80:                          //PB7
              :                       //PB7 related actions
            break;

            default:;                 //all other cases
    }                                 //end switch(new_PORTB)
  }                                   //end if new_PORTB
old_PORTB = new_PORTB;                //update PORTB
}

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

That completes our brief overview of the C programming language. In the next section, we discuss how to program the Microchip ATmega328 processor.

2.4 PROGRAMMING THE ATMEGA328

There are several different methods of programming the ATmega328.

1. **Onboard Arduino UNO R3 Programming:** In Chapter 1 we described how to program the ATmega328 while onboard the Arduino UNO R3 using the Arduino IDE. This tech-

nique allows access to the user-friendly features of the Arduino IDE and also retains the Arduino operating environment.

2. **Onboard Arduino UNO R3 using In System Programming (ISP):** This technique employs an ISP programmer (e.g., AVR Dragon) to program the ATmega328 while onboard the Arduino UNO R3. This technique allows access to the user-friendly hardware interface features of the Arduino UNO R3. However, when a program is loaded to the ATmega328 via the ISP programmer, the Arduino operating environment is overwritten. It is important to note that the Arduino operating environment can be rewritten to the ATmega328 if need be.

3. **ATmega328 using ISP:** This technique employs an ISP programmer (e.g., AVR Dragon) to program the ATmega328 via the Serial Peripheral Interface (SPI) system.

In this section we describe how to program the ATmega328 using ISP techniques.

2.4.1 ISP HARDWARE AND SOFTWARE TOOLS

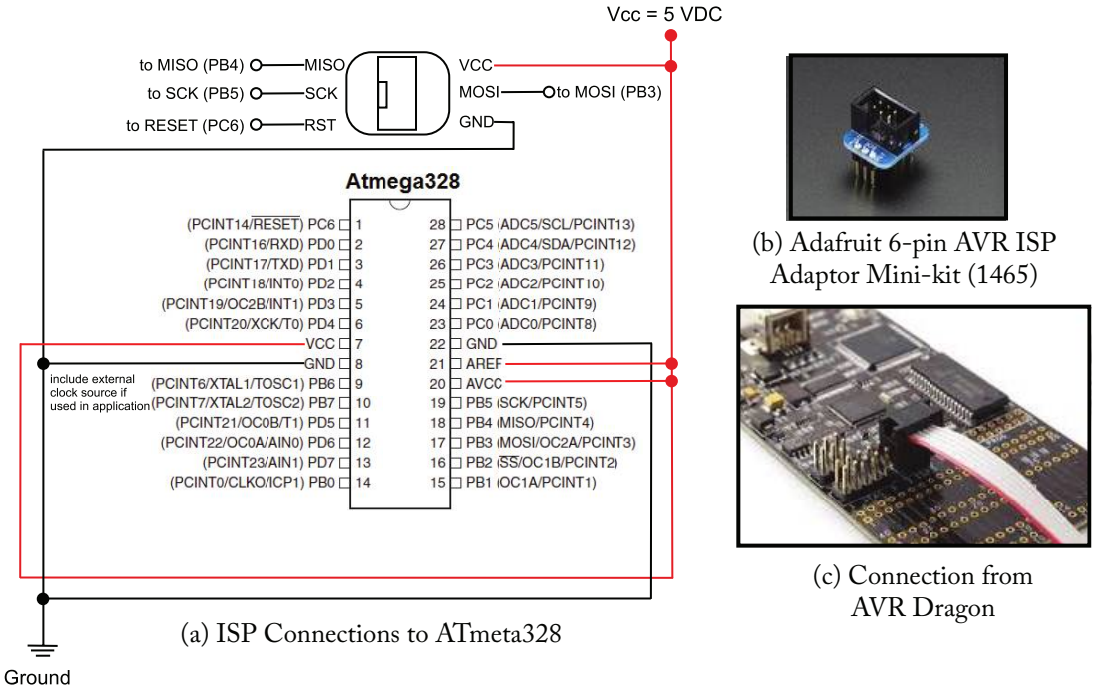
Programming the ATmega328 requires several hardware and software tools. For software tools a compiler and device programming support is required. Throughout the book we provide examples using both the ImageCraft JumpStart C for AVR compiler www.imagecraft.com and also the Atmel® Studio gcc compiler www.atmel.com. We use the Atmel® Studio software suite with the AVR Dragon programmer to download and program the microcontroller. The connection between the host computer and the AVR Dragon is shown in Figure 2.7 with detailed instructions.

2.4.2 IMAGECRAFT JUMPSTART C FOR AVR COMPILER DOWNLOAD, INSTALLATION, AND ATMEGA328 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(s) (testbench.c and testbench.h) into machine language (testbench.hex) for loading into the ATmega328. We use Microchip's Atmel® Studio to load the machine code into the ATmega328.

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

2. Create a new project (File $->$ New $->$ Project) and select "ImageCraft AVR Project." Press Next.



(a) ISP Connections to ATmeta328

(b) Adafruit 6-pin AVR ISP Adaptor Mini-kit (1465)

(c) Connection from AVR Dragon

Figure 2.7: Programming the ATmega328 with the AVR Dragon. (Image of Adafruit 1465 used with permission of Adafruit [www.adafruit.com]. Microchip AVR Dragon illustration used with permission of Microchip, Inc. [www.microchip.com].)

3. Provide a descriptive project title and browse to the desired folder location.
4. Select target as ATmega328. Details of the microcontroller and connected programming pod (e.g., AVR Dragon) will populate drop down windows. Press OK. A project with a "main.c" template will be created.
5. Open "main.c" and write your program.
6. With program writing complete, build project (Build – > Build). Correct any syntax errors and rebuild project. Repeat this process until no syntax errors remain. Upon a successful program build, a filename.hex and filename.elf are created.
7. Reference the next section to program the ATmega328 using ISP techniques.

2.4.3 ATMEL® STUDIO DOWNLOAD, INSTALLATION, AND ATMEGA328 PROGRAMMING

1. Download and install the latest version of Atmel® Studio.

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 2.7.
    - 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 ATmega328 target as shown.

4. Program Compiling and Programming

    - 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
        - ATmega328
        - Interface: ISP Apply
        - Insure target chip has 5 VDC applied

    Apply
        - Read signature, read target voltage
    - Memories: Flash: Program-Browse for desired "filename.hex" and press "Program."
    - Fuses: Ext. Crystal Osc. 8.0- MHz: Program

2.5 EXAMPLE: ATMEGA328 TESTBENCH

In this example, 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 ATmega328 subsystems working with various I/O devices. More importantly, the Testbench will serve as a template to develop your own applications.

2.5.1 HARDWARE CONFIGURATION

Provided in Figure 2.8 is the basic hardware configuration for the Testbench. PORTB is configured with eight tact (momentary) switches with accompanying debouncing hardware. PORTD 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.

Aside from the input hardware on PORTB and the output display hardware on PORTD of the controller, there are power (pins 7, 20, and 21) and ground (pins 8 and 22) 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. The RESET pin (pin 1) has 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 9 (XTAL1) and 10 (XTAL2) of the ATmega328. Hardware interface details of the Testbench are provided in "Arduino I: Getting Started!"

2.5.2 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 ATmega328 hardware 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. We also include the ImageCraft include file. Comment out the include file not in use.
- 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 ATmega328 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 PORTD.
- Function Definition

```
//*****************************************************************
//file name: testbench.c
//function: test bench for MICROCHIP AVR ATmega328 controller
```

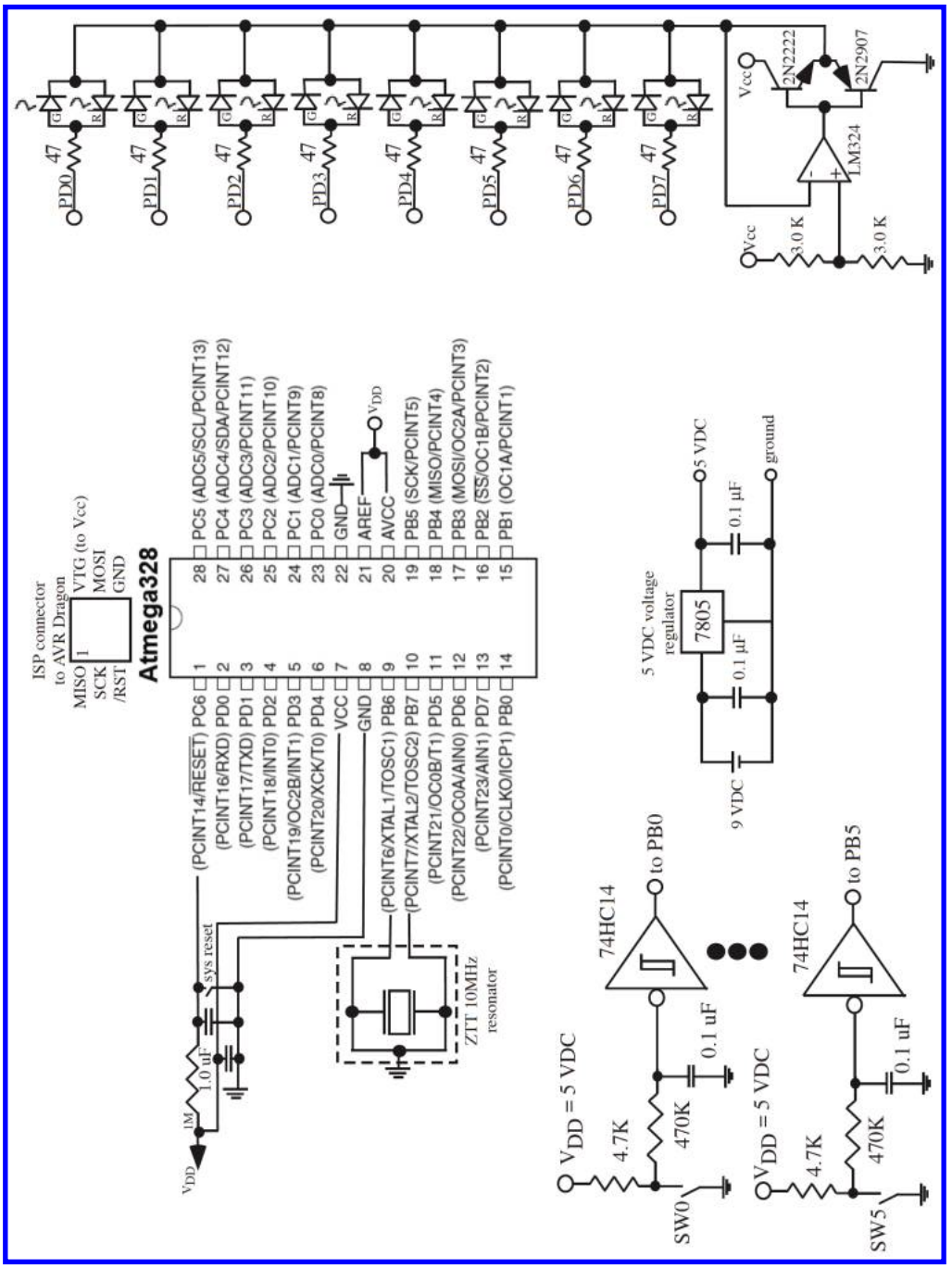


Figure 2.8: ATmega328 Testbench hardware.

```
//target controller: MICROCHIP ATmega328
//
//MICROCHIP AVR ATmega328 Controller Pin Assignments
//Chip Port Function I/O Source/Dest Asserted Notes
//Pin  1 PC6 /Reset pushbutton circuitry
//Pin  2 PD0 to tristate LED indicator
//Pin  3 PD1 to tristate LED indicator
//Pin  4 PD4 to tristate LED indicator
//Pin  5 PD3 to tristate LED indicator
//Pin  6 PD4 to tristate LED indicator
//Pin  7 VCC to 5 VDC
//Pin  8 GND to GND
//Pin  9 PB6 ZTT 10 MHz ceramic resonator
//Pin 10 PB7 ZTT 10 MHz ceramic resonator
//Pin 12 PD6 to tristate LED indicator
//Pin 13 PD7 to tristate LED indicator
//Pin 14 PB0 to active high RC debounced switch
//Pin 15 PB1 to active high RC debounced switch
//Pin 16 PB2 to active high RC debounced switch
//Pin 17 PB3 to active high RC debounced switch
//Pin 18 PB4 to active high RC debounced switch
//Pin 19 PB5 to active high RC debounced switch
//Pin 20 AVCC to 5 VDC
//Pin 21 AREF to 5 VDC
//Pin 22 GND to GND
//Pin 23 PC0
//Pin 24 PC1
//Pin 25 PC2
//Pin 26 PC3
//Pin 27 PC4
//Pin 28 PC5
//*****************************************************************

//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<iom328v.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;

          default:;                 //all other cases
          }                         //end switch(new_PORTB)
        }                           //end if new_PORTB

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

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

void initialize_ports(void)
{
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
}
//*****************************************************************
```

2.6 EXAMPLE: RAIN GAUGE INDICATOR

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 an upcoming chapter.

```
//*****************************************************************
//Rain Gauge
//*****************************************************************

//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<iom328v.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();
  }
}

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

void initialize_ports(void)
{
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

}

//*********************************************************************
//delay_100ms: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 10 MHz
//  - 100 ms delay requires 4M clock cycles
//  - nop requires 1 clock cycle to execute
//*********************************************************************

void delay_100ms(void)
{
unsigned int i,j;
```

```
for(i=0; i < 4000; 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 10 MHz
//  - 100 ms delay requires 4M 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();
  }
}

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

2.7 EXAMPLE: LOOP PRACTICE

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: loop practice
//*****************************************************************

//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<iom328v.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();
  }

}

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

2.8 SUMMARY

The goal of this chapter was to provide a tutorial on how to begin programming. We used a top-down design approach. We began with the "big picture" of the chapter followed by an overview of the ADE. We then discussed the basics of the C programming language. Only the most fundamental concepts were covered. We concluded with several examples.

2.9 REFERENCES

[1] S. F. Barrett and D. J. Pack. *Microcontroller Programming and Interfacing – Texas Instruments MSP430*, Synthesis Lectures on Digital Circuits and Systems, Morgan & Claypool Publishers, 2011. DOI: 10.2200/s00317ed1v01y201105dcs032. 30, 31

[2] ImageCraft Embedded Systems C Development Tools, Palo Alto, CA. www.imagecraft.com 28, 33

[3] Arduino homepage. www.arduino.cc

[4] *Microchip ATmega328 PB AVR Microcontroller with Core Independent Peripherals and Pico Power Technology DS40001906C*, Microchip Technology Incorporation, 2018. www.microchip.com

2.10 CHAPTER PROBLEMS

1. Describe the steps in writing a sketch and executing it on an Arduino UNO R3 processing board.
2. Describe the key portions of a C program.
3. Describe two different methods to program an Arduino processing board.
4. What is an include file?
5. What are the three pieces of code required for a program function?
6. Describe how to define a program constant.
7. Provide the C program statement to set PORTB pins 1 and 7 to logic one. Use bit-twiddling techniques.
8. Provide the C program statement to reset PORTB pins 1 and 7 to logic zero. Use bit-twiddling techniques.
9. What is the difference between a for and while loop?

10. When should a switch statement be used vs. the if-then statement construct?
11. What is the serial monitor feature used for in the Arduino Development Environment?
12. Why is the delay function provided in the chapter imprecise? Explain.
13. Can C programming commands be used within the Arduino Development Environment?
14. Can an entire C program be compiled and executed within the Arduino Development Environment?
15. What is the purpose of a "nop" command in assembly language?

CHAPTER 3

Analog to Digital Conversion (ADC)

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

- illustrate the analog-to-digital conversion process;
- assess the quality of analog-to-digital conversion using the metrics of sampling rate, quantization levels, number of bits used for encoding, and dynamic range;
- design signal conditioning circuits to interface sensors to analog-to-digital converters;
- implement signal conditioning circuits with operational amplifiers;
- describe the key registers used during an ATmega328 ADC;
- describe the steps to perform an ADC with the ATmega328;
- program the Arduino UNO R3 processing board to perform an ADC using the built-in features of the Arduino Development Environment;
- program the ATmega328 to perform an ADC in C; and
- describe the operation of a digital-to-analog converter (DAC).

3.1 OVERVIEW

A microcontroller is used to process information from the natural world, decide on a course of action based on the information collected, and then issue control signals to implement the decision. Since 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 digital form is required. An ADC system performs this task while a digital to analog converter (DAC) performs the conversion in the opposite direction. We will discuss both types of converters in this chapter. Most microcontrollers are equipped with an ADC subsystem, whereas DACs must be added as an external peripheral device to the controller.

In this chapter, we discuss the ADC process in some detail.[1] In the first section, we discuss the conversion process itself, followed by a presentation of the successive-approximation

[1]The sections on ADC theory were adapted with permission from *Microcontroller Fundamentals for Engineers and Scientists*, S. F. Barrett and D. J. Pack, Morgan & Claypool Publishers, 2006.

hardware implementation of the process. We then review the basic features of the ATmega328 ADC system followed by a system description and a discussion of key ADC registers. We conclude our discussion of the analog-to-digital converter with several illustrative code examples. We show how to program the ADC using the built-in features of the ADE and C. We conclude the chapter with a discussion of the DAC process. We also discuss the ADE built-in features that allow generation of an output analog signal via PWM techniques. Throughout the chapter, we provide detailed examples.

3.2 SAMPLING, QUANTIZATION, AND ENCODING

In this section, we provide an abbreviated discussion of the ADC process. This discussion was condensed from *Atmel® AVR Microcontroller Primer Programming and Interfacing*. The interested reader is referred to this text for additional details and examples in Barrett and Pack [1]. We present three important processes associated with the ADC: sampling, quantization, and encoding.

Sampling. We first start with the subject of sampling. Sampling is the process of taking "snapshots" of a signal over time. 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 retaining the capability to faithfully reconstruct the original signal from the samples. Intuitively, 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.

Sampling is important since 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. Harry Nyquist from Bell Laboratory studied the sampling process and derived a criterion that determines the minimum sampling rate 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. Engineers who work for telephone companies must deal with such issues. For further study on the Nyquist sampling rate, refer to Pack and Barrett listed in the References section.

When a signal is sampled a low pass anti-aliasing filter must be employed to insure the Nyquist sampling rate is not violated. In the example above, a low-pass filter with a cutoff frequency of 4 KHz would be used before the sampling circuitry for this purpose.

Quantization. Now that we understand the sampling process, let's move on to the second process of the ADC, quantization. Each digital system has a number of bits it uses as the basic unit to represent data. A bit is the most basic unit where single binary information, one or zero, is represented. A nibble is made up of four bits put together. A byte is eight bits.

We have tacitly avoided the discussion of the form of captured signal samples. When a 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 the quantization levels. Suppose you have a single bit to represent an incoming signal. You only have two different numbers, 0 and 1. You may say that you can distinguish only low from high. Suppose you have two bits. You can represent four different levels, 00, 01, 10, and 11. What if you have three bits? You now can represent eight different 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 one or zero, making the total possibilities eight. Similar discussion can lead us to conclude that given n bits, we have 2^n unique numbers or levels one can represent.

Figure 3.1 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 (pressure, temperature, etc.) with the help of transducers, such as microphones, angle sensors, and infrared sensors. The voltage signals are then conditioned to map their range with the input range of a digital system, typically 0–5 volts. In Figure 3.1, 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, the more quantization levels means the better mapping of an incoming signal to its true value. If we only had a single bit, we can only represent level 0 and level 1. Any analog signal value in between the range had to be mapped either as level 0 or level 1, not 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 accordingly.

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 eight bits and each sampled signal is quantized as one of 256 quantization levels. Consider the two sampled signals shown in Figure 3.1. 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. The quantization error is inversely proportional to the number of bits used to quantize the signal.

Encoding. 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, while the encoded sampled value for the second sample is 1100_0110. 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, shown in Figure 3.1.

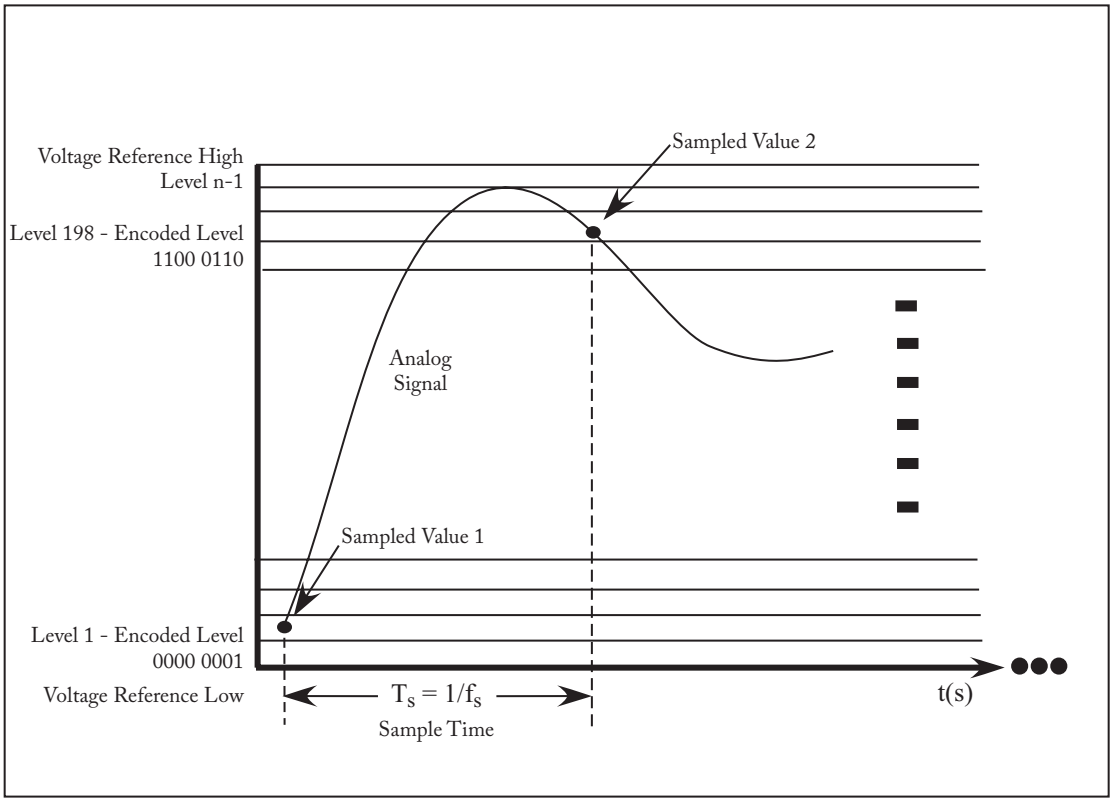


Figure 3.1: Sampling, quantization, and encoding.

3.2.1 RESOLUTION AND DATA RATE

Resolution. Resolution is a measure used to quantize an analog signal. In fact, resolution is nothing more than the voltage "distance" between two adjacent quantization levels we discussed earlier. Suppose again we have a range of 5 volts and one bit to represent an analog signal. The resolution in this case is 2.5 volts, a very poor resolution. You can imagine how your TV 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 volts 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 volts or 25% of the input range. Suppose you have 20 bits for quantization levels. The resolution now becomes 4.77×10^{-6} volts, 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 fixed voltage range, the distance between adjacent levels decreases, reducing the quantization error of a sampled signal. As the number grows, 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. You now should understand the technical background when you watch high definition television broadcasting. In general, resolution may be defined as:

$$\text{resolution} = (\text{voltage span})/2^b = (V_{ref\ high} - V_{ref\ low})/2^b$$

for the ATmega328, the resolution is:

$$\text{resolution} = (5 - 0)/2^{10} = 4.88\ \text{mV}.$$

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. In particular, 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 (f_s) and you are using 10 bits for the quantization levels for each sampled signal.[2] This means the voice conversation will be sampled every 125 μs (T_s) due to the reciprocal relationship between (f_s) and (T_s). 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 $\times$ 8000 $\times$ 10 or eight billion bits per every second! You will need some major computing power to meet the requirement for processing and storage of the data. For such reasons, when designers make decisions on the number of bits used for the quantization levels and the sampling rate, they must consider the computational burden the selection will produce on the computational capabilities of a digital system vs. the required system resolution.

Dynamic range. 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 (dB), which is the strength of a signal with respect to a reference signal. The greater the dB number, the stronger the signal is compared to 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, you will find 8–12 bits used in commercial analog-to-digital converters, translating the dynamic range from 20 $\log\ 2^8$ dB to 20 $\log\ 2^{12}$ dB.

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

3.3 ANALOG-TO-DIGITAL CONVERSION (ADC) PROCESS

The goal of the ADC process 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 ADC 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 personal 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 and from electrical signals. It is important to carefully design the interface between transducers and the microcontroller to insure proper operation. A poorly designed interface could result in improper embedded system operation or failure. Specific input and output transducer interface techniques are discussed in *Arduino I: Getting Started.*

3.3.1 TRANSDUCER INTERFACE DESIGN (TID) CIRCUIT

In addition to transducers, we also need a signal conditioning circuitry before we apply the ADC. 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 range of the analog-to-digital converter which is typically 0–5 VDC. Figure 3.2 shows the transducer interface circuit using an input transducer. This process assumes a linear input transducer.

The output of the input transducer is first scaled by constant K. As an example, in the figure, we use a microphone as the input transducer whose output ranges from −5 to +5 VDC. The input to the analog-to-digital converter ranges from 0–5 VDC. The scalar multiplier 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 VDC to +2.5 VDC instead of 0–5 VDC. The second portion of the circuit, the bias stage, shifts the range by 2.5 VDC, thereby completing the correct mapping. Actual implementation of the TID circuit components is accomplished using operational amplifiers.

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

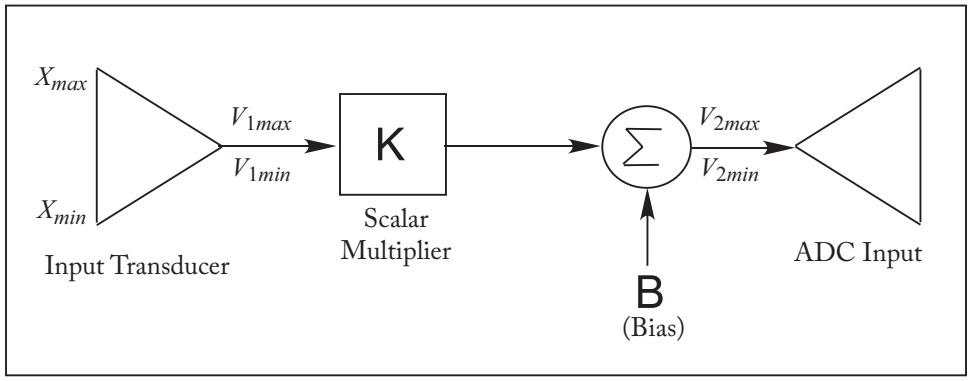


Figure 3.2: A block diagram of the signal conditioning for an analog-to-digital converter. The range of the sensor voltage output is mapped to the analog-to-digital converter input voltage range. The scalar multiplier maps the magnitudes of the two ranges and the bias voltage is used to align two limits.

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

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

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 converter [USAFA].

Similarly, the 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 converter.

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 (usually 5 VDC and 0 VDC for a microcontroller). 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.

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 volts for maximum rated light intensity and −2.50 VDC 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.

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

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

$$5.0\ V = (0\ V \times K) + B$$

$$0\ V = (-2.50\ V \times K) + B.$$

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

3.3.2 OPERATIONAL AMPLIFIERS

In the previous section, we discussed the transducer interface design (TID) process. Going through this design 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 the References section at the end of the chapter for pointers to some excellent texts on this topic.

3.3.2.1 The Ideal Operational Amplifier

A generic ideal operational amplifier is illustrated in Figure 3.3. 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 to the supply voltages $\pm V_{cc}$. This will clip the output, and hence distort the signal, at levels slightly less than $\pm V_{cc}$. 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 3.4 (Faulkenberry [7]).

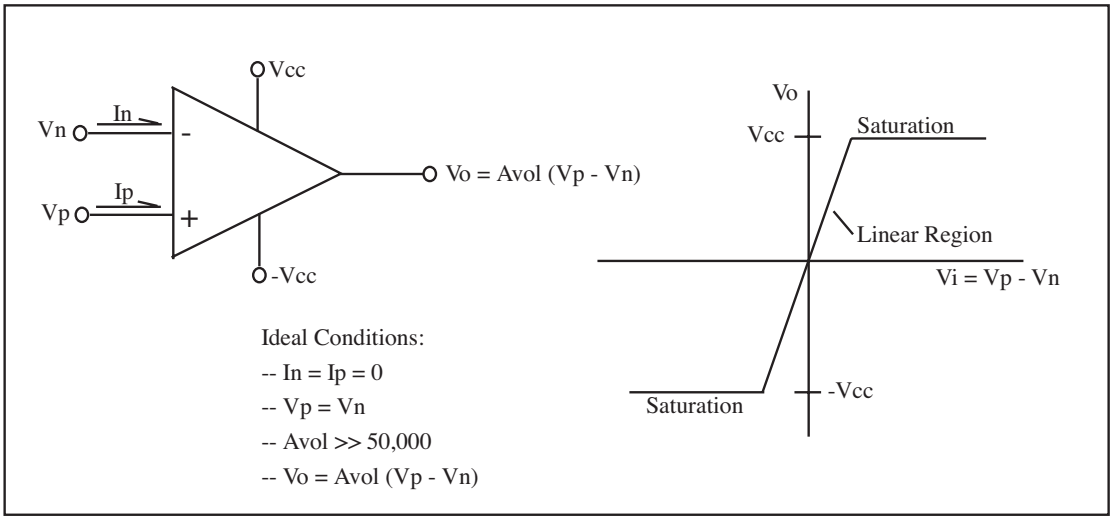


Figure 3.3: Ideal operational amplifier characteristics.

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 Vn for the circuit.
- Apply ideal op amp characteristics to the node equation.
- Solve the node equation for Vo.

As an example, we provide the analysis of the non-inverting amplifier circuit in Figure 3.5. This same analysis technique may be applied to all of the circuits in Figure 3.4 to arrive at the equations for Vout 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 provided in Figure 3.6 implements these values of K and B. The first stage provides an amplification of -2 due to the use of the inverting amplifier configuration. In the second stage, a summing 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.

3.4 ADC CONVERSION TECHNOLOGIES

The ATmega328 uses a successive-approximation converter technique to convert an analog sample into a 10-bit digital representation. In this section, we will discuss this type of conversion

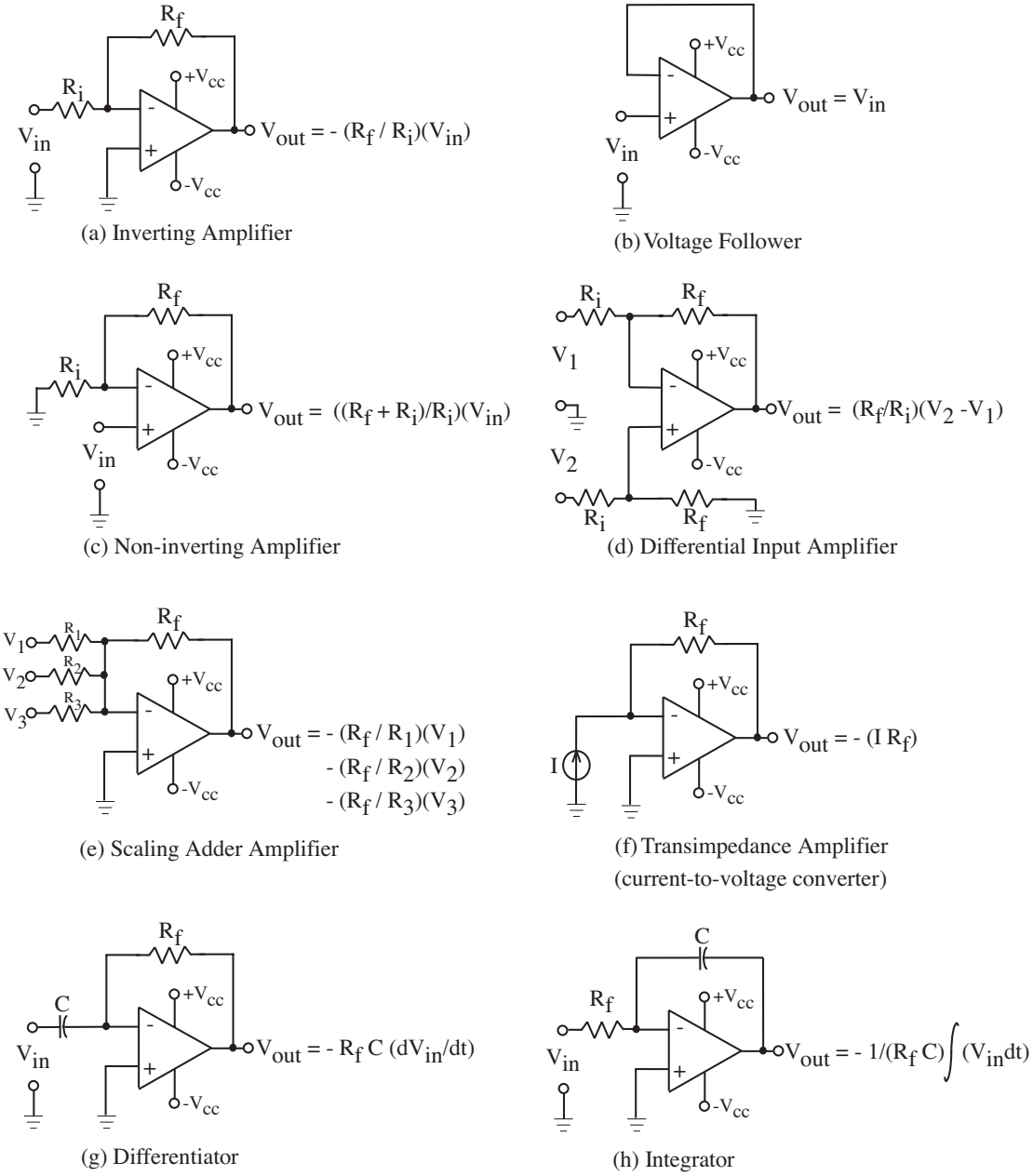


Figure 3.4: Classic operational amplifier configurations. Adapted from Faulkenberry [7].

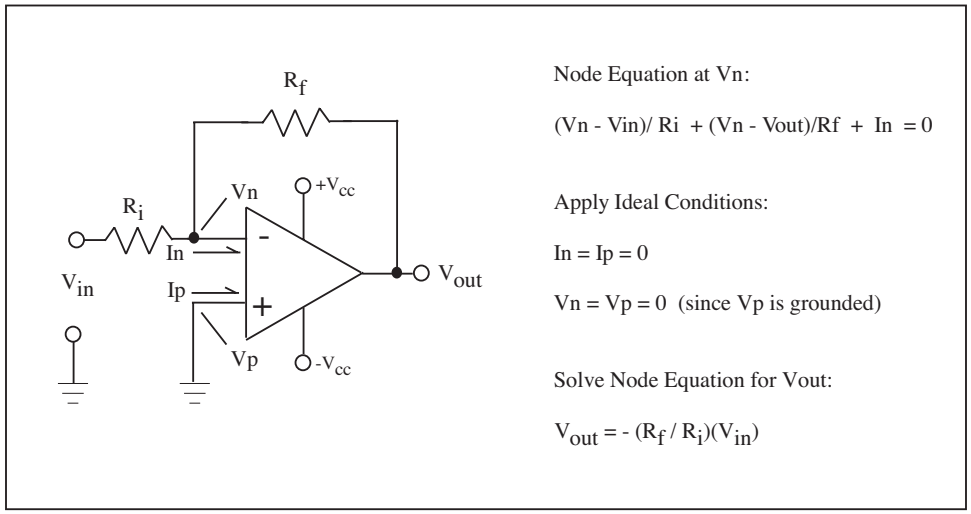


Figure 3.5: Operational amplifier analysis for the non-inverting amplifier. Adapted from Faulkenberry [7].

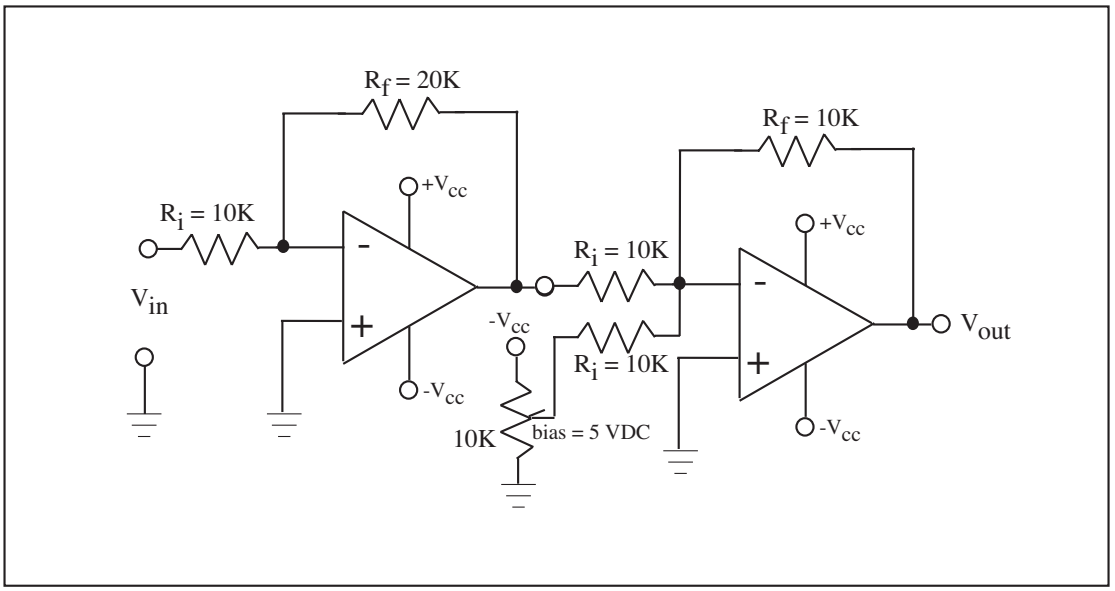


Figure 3.6: Operational amplifier implementation of the transducer interface design (TID) example circuit.

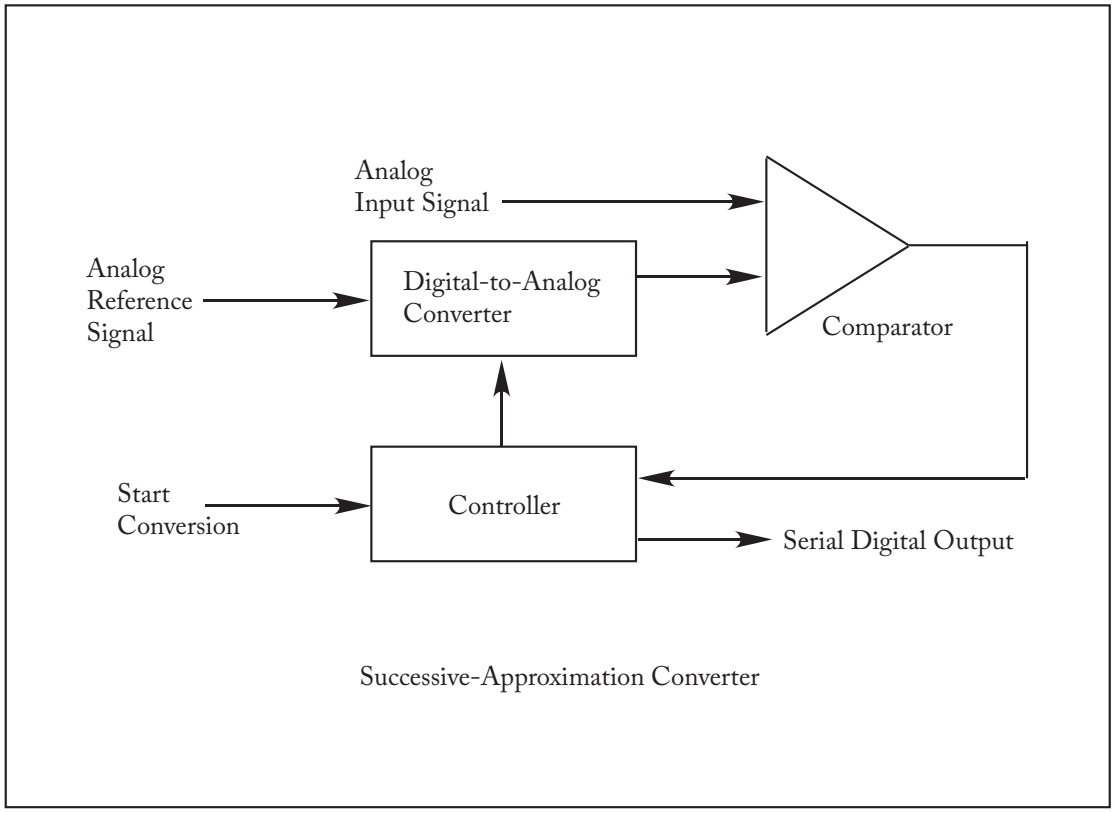


Figure 3.7: Successive-approximation ADC.

process. For a review of other converter techniques, the interested reader is referred to "Atmel® AVR Microcontroller Primer: Programming and Interfacing." In certain applications, you are required to use converter technologies external to the microcontroller.

The successive-approximation technique uses a DAC, a controller, and a comparator to perform the ADC process. Starting from the most significant bit down to the least significant bit, 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 most significant bit. The process continues until decisions are made for all available bits. Figure 3.7 shows the architecture of this type of converter. The advantage of this technique is that the conversion time is uniform for any input, but the disadvantage of the technology is the use of complex hardware for implementation.

3.5 THE MICROCHIP ATMEGA328 ADC SYSTEM

The Microchip ATmega328 microcontroller is equipped with a flexible and powerful ADC system. It has the following features (Microchip [2]):

- 10-bit resolution,
- ±2 least significant bit (LSB) absolute accuracy,
- 13 ADC clock cycle conversion time,
- six multiplexed, single-ended, input channels,
- selectable right or left result justification, and
- 0 to V_{cc} ADC input voltage range.

Let's discuss each feature in turn. 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 ATmega processor 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 [Microchip [2]].

It requires 13 analog-to-digital clock cycles to perform an ADC conversion. The ADC system may be run at a slower clock frequency 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). A slower ADC clock results in improved ADC accuracy at higher controller clock speeds.

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 a multiple input analog multiplexer. The analog input for conversion is selected using the MUX[3:0] 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 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 volts. If this is not the case, external circuitry must be used to insure the analog input voltage is within these prescribed bounds, as discussed earlier in the chapter.

3.5.1 BLOCK DIAGRAM

The block diagram for the ATmega328 ADC conversion system is provided in Figure 3.8. The left edge of the diagram provides the external microcontroller pins to gain access to the ADC. For the ATmega328, the six analog input channels are provided at pins ADC[5:0]. 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.

3.5.2 ATMEGA328 ADC REGISTERS

The key registers for the ATmega328 ADC system are shown in Figure 3.9. 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 setting already. We will discuss each register in turn [www.microchip.com].

3.5.2.1 ATmega328 ADC Multiplexer Selection Register (ADMUX)

As previously discussed, the ADMUX register contains the ADLAR bit to select left or right justification and the MUX[3:0] 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[3:0] bits. For example, to convert channel ADC7, "0111" 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
- REFS[1:1] = 11: Internal 1.1 VDC voltage reference with an external capacitor at the AREF pin

3.5.2.2 ATmega328 ADC Control and Status Register A (ADCSRA)

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 a logic one. The ADC Start Conversion (ADSC) bit is also contained in the ADCSRA register. Setting this bit to logic one initiates an ADC. The ADCSRA register also contains the ADC Interrupt flag (ADIF) bit.

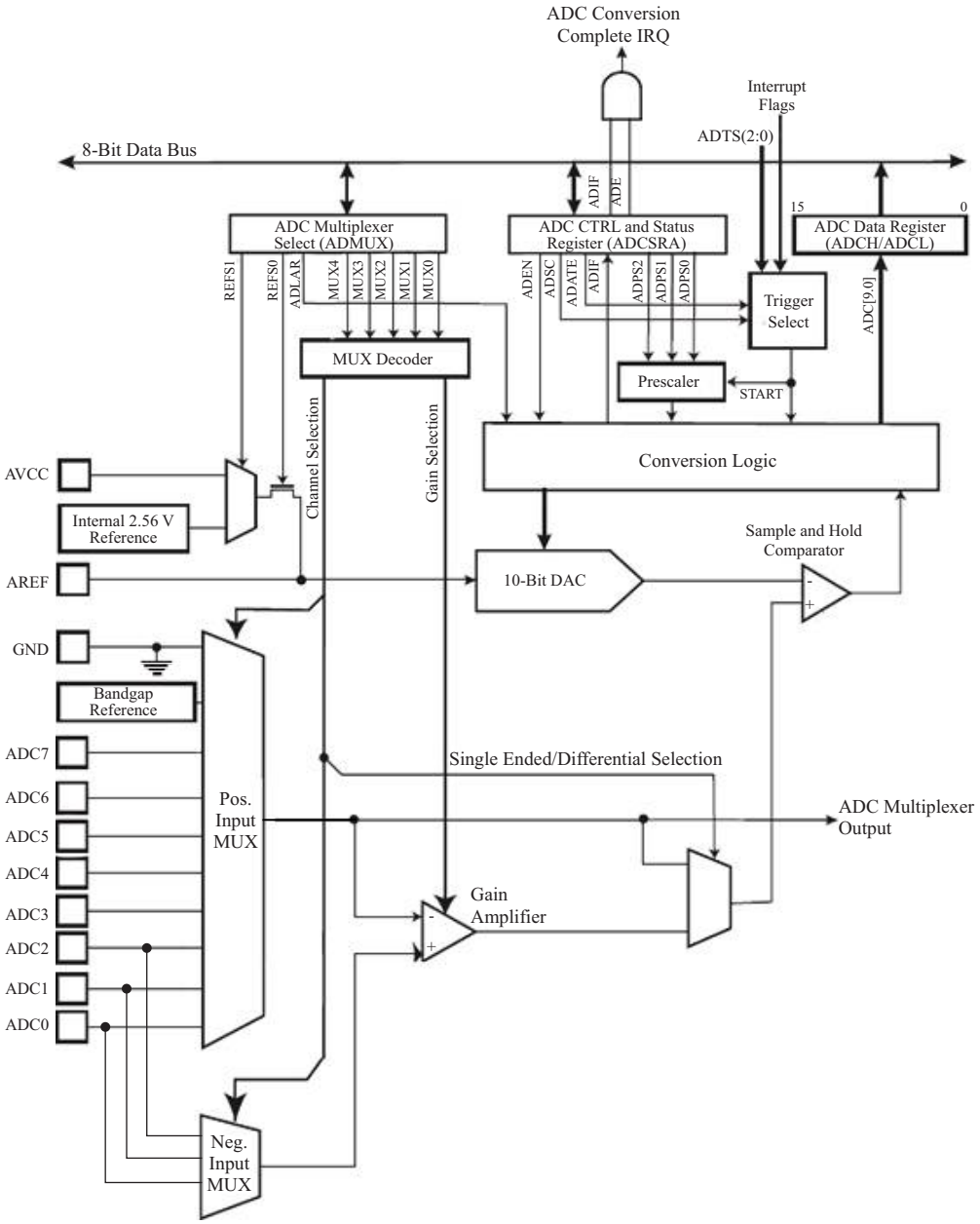


Figure 3.8: Microchip AVR ATmega328 ADC block diagram. (Figure used with permission of Microchip, Inc. [www.microchip.com].)

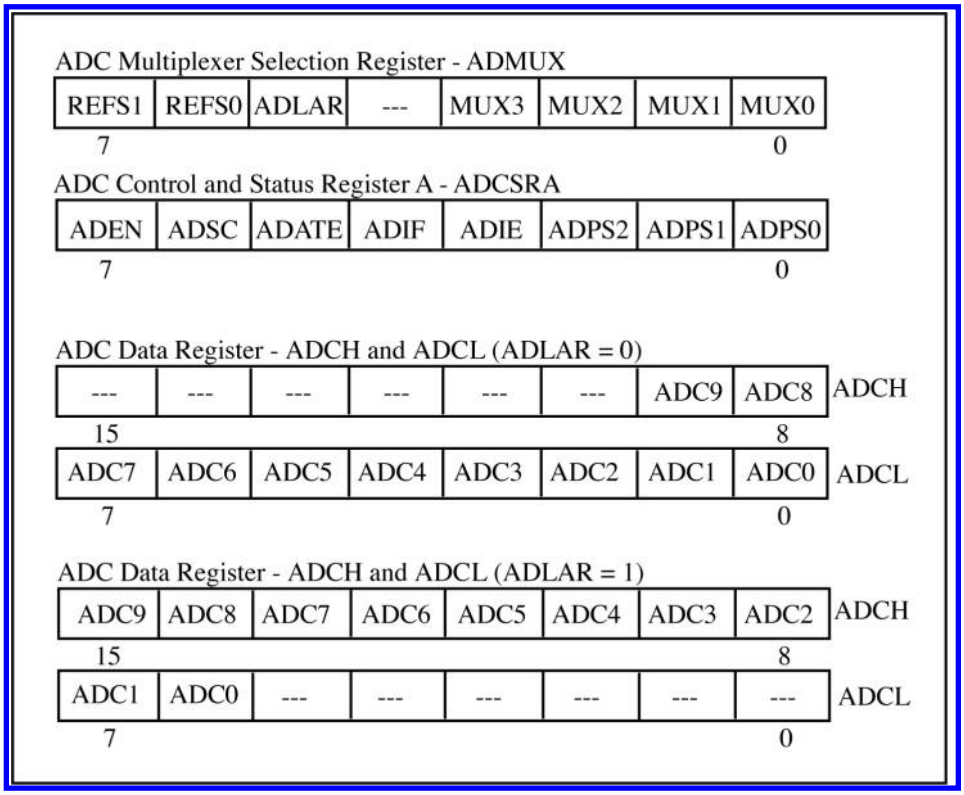


Figure 3.9: ATmega328 ADC Registers. Adapted from Microchip [www.microchip.com].

This bit sets to logic one when the ADC is complete. The ADIF bit is reset by writing a logic one to this bit.

The ADC Prescaler Select (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

3.5.2.3 ATmega328 ADC Data Registers (ADCH, 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.

3.6 PROGRAMMING THE ADC USING THE ARDUINO DEVELOPMENT ENVIRONMENT

The ADE has the built-in function analogRead to perform an ADC conversion. The format for the analogRead function is:

```
unsigned int  return_value;


return_value = analogRead(analog_pin_read);
```

The function returns an unsigned integer value from 0–1023, corresponding to the voltage span from 0–5 VDC.

3.7 PROGRAMMING THE ADC IN C

Provided below are two functions to operate the ATmega328 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 one 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 eight bit ADCL and ADCH result registers are concatenated into a 16-bit unsigned integer variable and returned to the function call.

```
//*****************************************************************
//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
while (!(ADCSRA & 0x10));               //Check if conversation ready
ADCSRA |= 0x10;                         //Clear conv rdy flag - set 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 right 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;
                                    //weighted binary voltage
unsigned int binary_weighted_voltage_high;

ADMUX = channel;                    //Select channel
ADCSRA |= 0x43;                     //Start conversion
                                    //Set ADC module prescalar to 8
                                    //critical for accurate ADC
while (!(ADCSRA & 0x10));           //Check if conversion is ready
ADCSRA |= 0x10;                     //Clear Conv rdy flag - set 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
}

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

3.8 EXAMPLE: ADC RAIN GAGE INDICATOR WITH THE ARDUINO UNO R3

In this example, we construct a rain gage-type level display using small light-emitting diodes. The rain gage indicator consists of a panel of eight light-emitting diodes. The gage may be constructed from individual diodes or from an LED bar containing eight elements. Whichever style is chosen, the interface requirements between the processor and the LEDs are the same.

The requirement for this project is to use the ADC to illuminate up to eight LEDs based on the input voltage. A 10-K Ohm trimmer potentiometer is connected to the ADC channel to vary the input voltage. We first provide a solution using the ADE with the Arduino UNO R3 processing board. Then a solution employing the ATmega328 programmed in C is provided.

3.8.1 ADC RAIN GAGE INDICATOR USING THE ARDUINO DEVELOPMENT ENVIRONMENT

The circuit configuration employing the Arduino UNO R3 processing board is provided in Figure 3.10. The DIGITAL pins of the microcontroller are used to communicate with the LED interface circuit. The operation of the LED interface circuit is provided in *Arduino I: Getting Started*.

The sketch to implement the project requirements is provided below. As in previous examples, we define the Arduino UNO R3 pins, set them for output via the setup() function, and write the loop() function. In this example, the loop() function senses the voltage from the 10-K trimmer potentiometer and illuminates a series of LEDs corresponding to the sensed voltage levels.

```
//*****************************************************************
#define trim_pot  A0                  //analog input pin

                                      //digital output pins
                                      //LED indicators 0 - 7
#define LED0       0                  //digital pin
#define LED1       1                  //digital pin
#define LED2       2                  //digital pin
#define LED3       3                  //digital pin
#define LED4       4                  //digital pin
#define LED5       5                  //digital pin
```

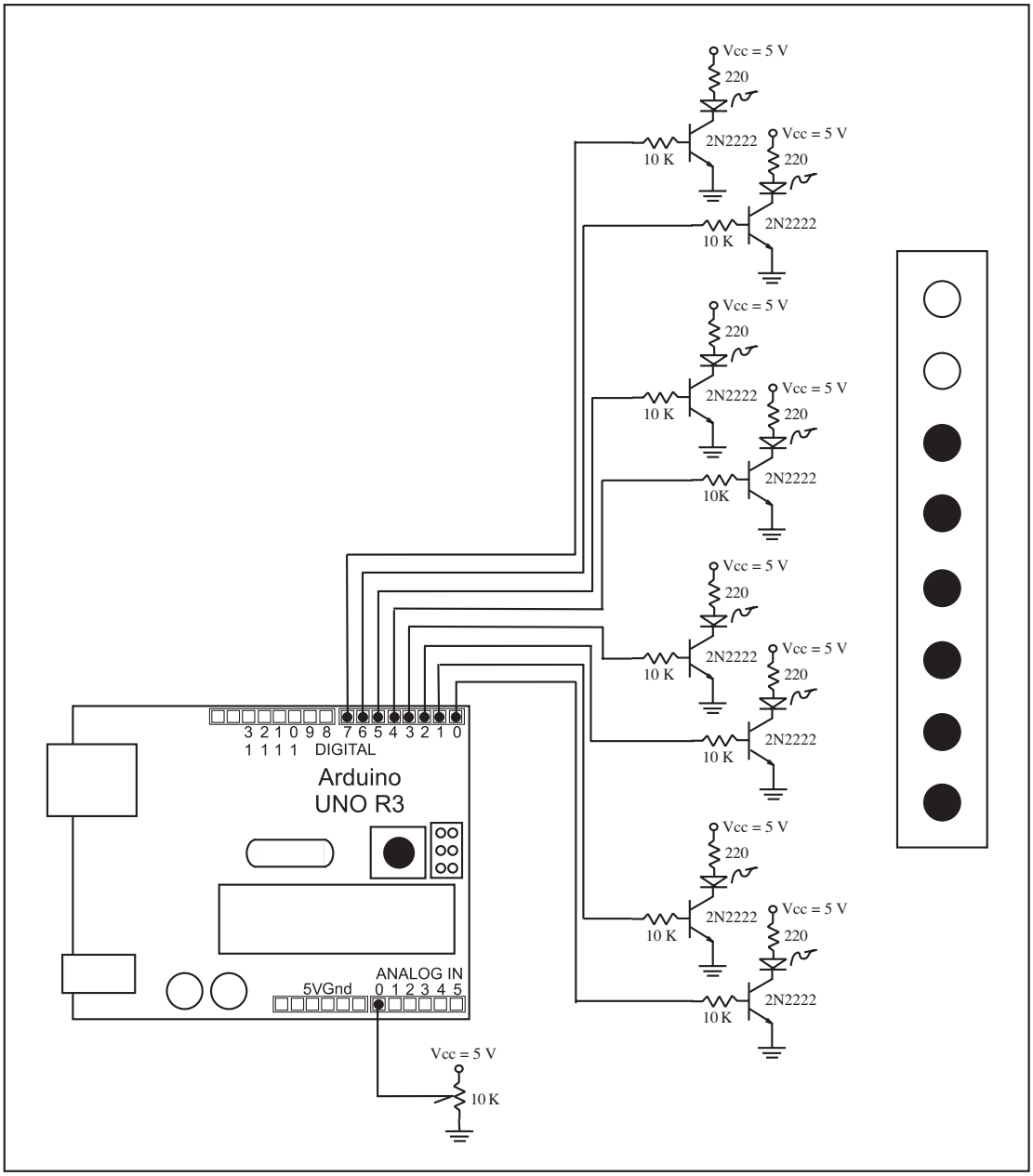


Figure 3.10: ADC with rain gage level indicator.

```
#define LED6       6                  //digital pin
#define LED7       7                  //digital pin


int trim_pot_reading;                 //declare variable for trim pot

void setup()
  {
  pinMode(LED0, OUTPUT);              //config pin 0 for digital out
  pinMode(LED1, OUTPUT);              //config pin 1 for digital out
  pinMode(LED2, OUTPUT);              //config pin 2 for digital out
  pinMode(LED3, OUTPUT);              //config pin 3 for digital out
  pinMode(LED4, OUTPUT);              //config pin 4 for digital out
  pinMode(LED5, OUTPUT);              //config pin 5 for digital out
  pinMode(LED6, OUTPUT);              //config pin 6 for digital out
  pinMode(LED7, OUTPUT);              //config pin 7 for digital out
  }

void loop()
  {
                                      //read analog out from trim pot
   trim_pot_reading = analogRead(trim_pot);

   if(trim_pot_reading < 128)
     {
     digitalWrite(LED0, HIGH);  digitalWrite(LED1, LOW);
     digitalWrite(LED2, LOW);   digitalWrite(LED3, LOW);
     digitalWrite(LED4, LOW);   digitalWrite(LED5, LOW);
     digitalWrite(LED6, LOW);   digitalWrite(LED7, LOW);
     }
   else if(trim_pot_reading < 256)
     {
     digitalWrite(LED0, HIGH);   digitalWrite(LED1, HIGH);
     digitalWrite(LED2, LOW);    digitalWrite(LED3, LOW);
     digitalWrite(LED4, LOW);    digitalWrite(LED5, LOW);
     digitalWrite(LED6, LOW);    digitalWrite(LED7, LOW);
     }
   else if(trim_pot_reading < 384)
     {
```

```
    digitalWrite(LED0, HIGH);   digitalWrite(LED1, HIGH);
    digitalWrite(LED2, HIGH);   digitalWrite(LED3, LOW);
    digitalWrite(LED4, LOW);    digitalWrite(LED5, LOW);
    digitalWrite(LED6, LOW);    digitalWrite(LED7, LOW);
    }
  else if(trim_pot_reading < 512)
    {
    digitalWrite(LED0, HIGH);   digitalWrite(LED1, HIGH);
    digitalWrite(LED2, HIGH);   digitalWrite(LED3, HIGH);
    digitalWrite(LED4, LOW);    digitalWrite(LED5, LOW);
    digitalWrite(LED6, LOW);    digitalWrite(LED7, LOW);
    }
  else if(trim_pot_reading < 640)
    {
    digitalWrite(LED0, HIGH);   digitalWrite(LED1, HIGH);
    digitalWrite(LED2, HIGH);   digitalWrite(LED3, HIGH);
    digitalWrite(LED4, HIGH);   digitalWrite(LED5, LOW);
    digitalWrite(LED6, LOW);    digitalWrite(LED7, LOW);
    }
  else if(trim_pot_reading < 768)
    {
    digitalWrite(LED0, HIGH);   digitalWrite(LED1, HIGH);
    digitalWrite(LED2, HIGH);   digitalWrite(LED3, HIGH);
    digitalWrite(LED4, HIGH);   digitalWrite(LED5, HIGH);
    digitalWrite(LED6, LOW);    digitalWrite(LED7, LOW);
    }
  else if(trim_pot_reading < 896)
    {
    digitalWrite(LED0, HIGH);   digitalWrite(LED1, HIGH);
    digitalWrite(LED2, HIGH);   digitalWrite(LED3, HIGH);
    digitalWrite(LED4, HIGH);   digitalWrite(LED5, HIGH);
    digitalWrite(LED6, HIGH);   digitalWrite(LED7, LOW);
    }
  else
    {
    digitalWrite(LED0, HIGH);   digitalWrite(LED1, HIGH);
    digitalWrite(LED2, HIGH);   digitalWrite(LED3, HIGH);
    digitalWrite(LED4, HIGH);   digitalWrite(LED5, HIGH);
    digitalWrite(LED6, HIGH);   digitalWrite(LED7, HIGH);
```

```
    }
delay(500);                          //delay 500 ms
}

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

3.8.2 ADC RAIN GAGE INDICATOR IN C

We now implement the rain gage indicator with the control algorithm (sketch) programmed in C for the ATmega328. The circuit configuration employing the ATmega328 is provided in Figure 3.11. PORT D of the microcontroller is used to communicate with the LED interface circuit. The LED interface circuit is described in *Arduino I: Getting Started*. In this example, we extend the requirements of the project.

- Write a function to display an incrementing binary count from 0–255 on the LEDs.
- Write a function to display a decrementing binary count from 255–0 on the LEDs.
- Use the ADC to illuminate up to eight LEDs based on the input voltage. A 10-K trimmer potentiometer is connected to the ADC channel to vary the input voltage.

The project algorithm was written by Anthony (Tony) Kunkel, MSEE and Geoff Luke, Ph.D., at the University of Wyoming for an Industrial Controls class assignment. A 30-ms delay is provided between PORTD LED display updates. This prevents the display from looking as a series of LEDs that are always illuminated.

Note: The delay function provided in this example is not very accurate. It is based on counting the number of assembly language no operation (NOPs). If a single NOP requires a single clock cycle to execute. A rough (and inaccurate) estimation of expended time may be calculated. The actual delay depends on the specific resolution of clock source used with the microcontroller, the clock frequency of the clock source and the specific compiler used to implement the code. In this specific example an accurate delay is not required since we are simply trying to slow down the code execution time so the LED changes may be observed. In the next chapter we provide a more accurate method of providing a time delay based on counting interrupts.

```
//*************************************************************
//Tony Kunkel and Geoff Luke
//University of Wyoming
//*************************************************************
//This program calls four functions:
//First: Display an incrementing binary count on PORTD from 0-255
//Second: Display a decrementing count on PORTD from 255-0
//Third: Display rain gauge info on PORTD
```

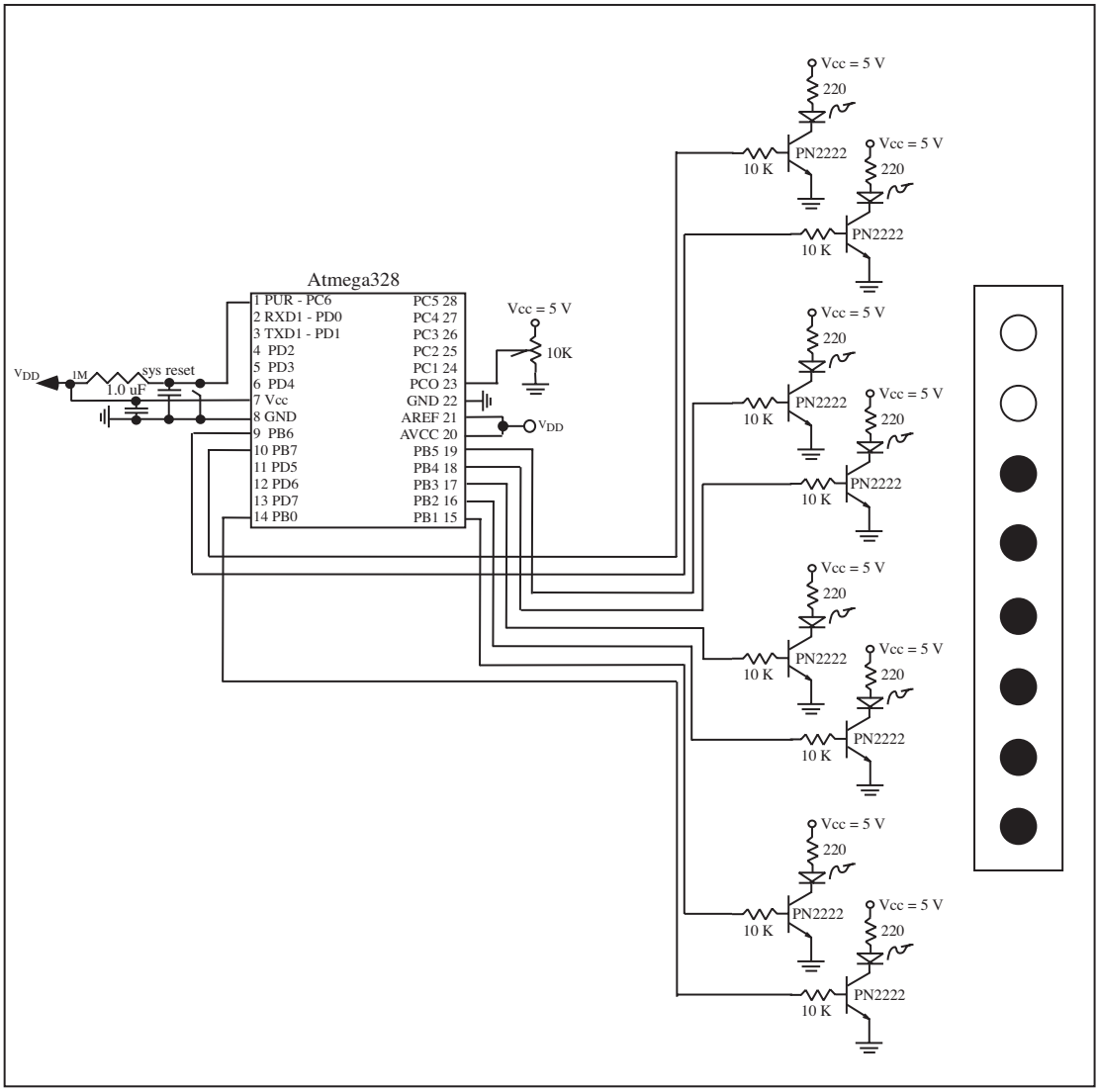


Figure 3.11: ADC with rain gage level indicator.

```
//Fourth: Delay whenever PORTD is updated
//*****************************************************************

//MICROCHIP register definitions for ATmega328
#include<iom328pv.h>

//function prototypes
void display_increment(void);  //func displays increment to PORTD
void display_decrement(void);  //func displays decrement to PORTD
void rain_gage(void)
void InitADC(void);            //initialize ADC converter
unsigned int ReadADC();        //read specified ADC channel
void delay(void);              //function to delay

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

int main(void)
{
display_increment();           //display incrementing binary on
                               //PORTD from 0-255
delay();                       //delay
display_decrement();           //display decrementing binary on
                               //PORTD from 255-0
delay();                       //delay
InitADC();

while(1)
  {
  rain_gage();                 //display gage info on PORTD
  delay();                     //delay
  }
return 0;
}



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

```
//delay
//Note: The delay function provided in this example is not very
//accurate.  It is based on counting the number of assembly language
//no operation (NOPs).  If a single NOP requires a single clock cycle
//to execute.  A rough (and inaccurate) estimation of expended time
//may be calculated.  The actual delay depends on the specific
//resolution of clock source used with the microcontroller, the
//clock frequency of the clock source and the specific compiler used
//to implement the code.  In this specific example an accurate delay
//is not required since we are simply trying to slow down the code
//execution time so the LED changes may be observed.
//*****************************************************************

void delay(void)
{
int i, k;

for(i=0; i<400; i++)
  {
  for(k=0; k<300; k++)
      {
      asm("nop");            //assembly nop, requires 2 cycles
      }
  }
}

//*****************************************************************
//Displays incrementing binary count from 0 to 255
//*****************************************************************

void display_increment(void)
{
int i;
unsigned char j = 0x00;

DDRD = 0xFF;                 //set PORTD to output

for(i=0; i<255; i++)
  {
```

```
  j++;                          //increment j
  PORTD = j;                    //assign j to data port
  delay();                      //wait
  }
}

//*****************************************************************
//Displays decrementing binary count from 255 to 0
//*****************************************************************

void display_decrement(void)
{
int i;
unsigned char j = 0xFF;

DDRD = 0xFF;                    //set PORTD to output

for(i=0; i<256; i++)
  {
  j=(j-0x01);                   //decrement j by one
  PORTD = j;                    //assign char j to data port
  delay();                      //wait
  }
}


//*****************************************************************
//Initializes ADC
//*****************************************************************

void InitADC(void)
{
ADMUX = 0;                      //Select channel 0
ADCSRA = 0xC3;                  //Enable ADC & start 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 right 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 is ready
ADCSRA |= 0x10;                     //Clear Conv rdy flag - set bit
binary_weighted_voltage_low = ADCL; //Read 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
}

//*****************************************************************
//Displays voltage magnitude as LED level on PORTB
//*****************************************************************

void rain_gage(void)
{
unsigned int ADCValue;
```

```
ADCValue = readADC(0x00);
DDRD = 0xFF;                    //set PORTD to output

if(ADCValue < 128)
  {
  PORTD = 0x01;                 //illuminate LED at PORTD[0]
  }
else if(ADCValue < 256)
  {
  PORTD = 0x03;                 //illuminate LED at PORTD[1:0]
  }
else if(ADCValue < 384)
  {
  PORTD = 0x07;                 //illuminate LED at PORTD[2:0]
  }
else if(ADCValue < 512)
  {
  PORTD = 0x0F;                 //illuminate LED at PORTD[3:0]
  }
else if(ADCValue < 640)
  {
   PORTD = 0x1F;                //illuminate LED at PORTD[4:0]
  }
else if(ADCValue < 768)
  {
  PORTD = 0x3F;                 //illuminate LED at PORTD[5:0]
  }
else if(ADCValue < 896)
  {
  PORTD = 0x7F;                 //illuminate LED at PORTD[6:0]
  }
else
  {
  PORTD = 0xFF;                 //illuminate LED at PORTD[7:0]
  }
}

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

Example: In Chapter 7 an ADC voltmeter example is provided. In the example, ADC readings are provided to an LCD digit-by-digit for display.

3.9 ONE-BIT ADC – THRESHOLD DETECTOR

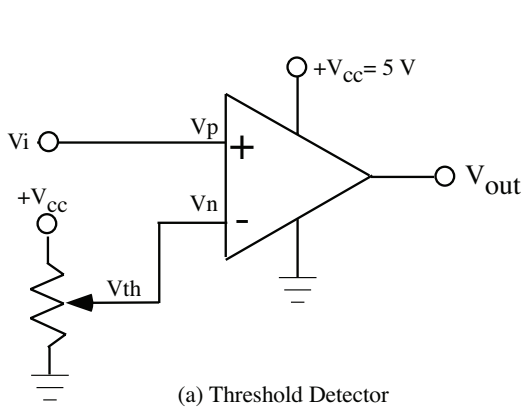
A threshold detector circuit or comparator configuration contains an operational amplifier employed in the open-loop configuration. That is, no feedback is provided from the output back to the input to limit gain. A threshold level is applied to one input of the op amp. This serves as a comparison reference for the signal applied to the other input. The two inputs are constantly compared to one another. When the input signal is greater than the set threshold value, the op amp will saturate to a value slightly less than $+V_{cc}$, as shown in Figure 3.12a. When the input signal falls below the threshold the op amp will saturate at a voltage slightly greater than V_{cc}. If a single-sided op amp is used in the circuit (e.g., LM324), the V_{cc} supply pin may be connected to ground. In this configuration, the op map provides for a one-bit ADC circuit.

A bank of threshold detectors may be used to construct a multi-channel threshold detector, as shown in Figure 3.12c. This provides a flash converter type ADC. It is a hardware version of a rain gage indicator. A 14-channel version for use in a laboratory instrumentation project is provided in Barrett and Sundberg [10].

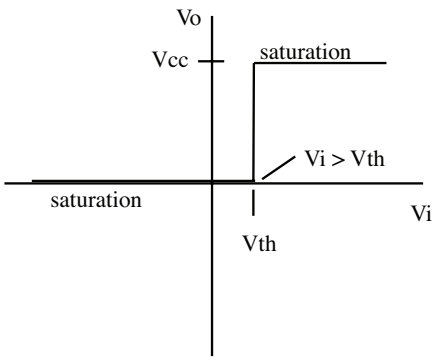
3.10 DIGITAL-TO-ANALOG CONVERSION (DAC)

Once a signal is acquired to a digital system with the help of the ADC process and has been processed, frequently the processed signal is 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. The most commonly used technique to convert digital signals to analog signals is the summation method shown in Figure 3.13.

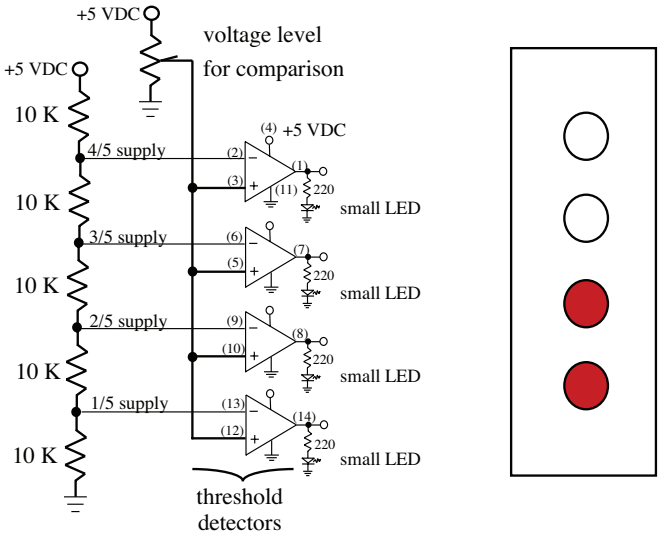
With the summation method of DAC, a digital signal, represented by a set of ones and zeros, enters the DAC from the most significant bit to the least significant bit. For each bit, a comparator checks its logic state, high or low, to produce a clean digital bit, represented by a voltage level. Typically, in a microcontroller context, the voltage level is +5 or 0 volts to represent logic one or logic zero, respectively. The voltage is then multiplied by a scalar value based on its significant position of the digital signal, as shown in Figure 3.13. Once all bits for the signal have been processed, the resulting voltage levels are summed together 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.



(a) Threshold Detector



(b) Transfer Characteristic for Threshold Detector



(c) 4-Channel Threshold Detector

Figure 3.12: One-bit ADC threshold detector.

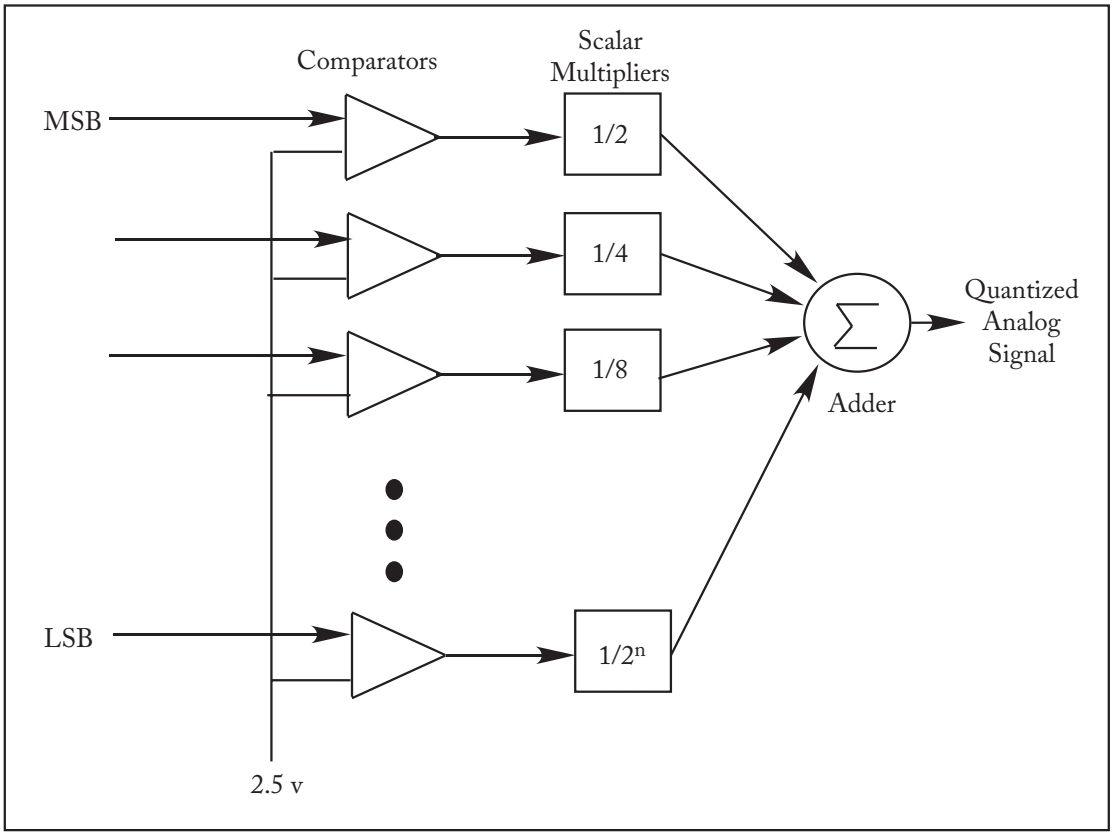


Figure 3.13: A summation method to convert a digital signal into a quantized analog signal. Comparators are used to clean up incoming signals and the resulting values are multiplied by a scalar multiplier and the results are added to generate the output signal. For the final analog signal, the quantized analog signal should be connected to a low-pass filter followed by a transducer interface circuit.

3.10.1 DAC WITH THE ARDUINO DEVELOPMENT ENVIRONMENT

The analogWrite command within the ADE issues a signal from 0–5 VDC by sending a constant from 0–255 using PWM techniques. This signal, when properly filtered, serves as a DC signal.

The form of the analogWrite command is the following:

analogWrite(output pin, value);

3.10.2 DAC WITH EXTERNAL CONVERTERS

A microcontroller can be equipped with a wide variety of DAC configurations including:

- single-channel, 8-bit DAC connected via a parallel port (e.g., Motorola MC1408P8, Texas Instruments DAC0808);
- quad channel, 8-bit DAC connected via a parallel port (e.g., Analog Devices AD7305);
- quad channel, 8-bit DAC connected via the SPI (e.g., Analog Devices AD7304); and
- octal channel, 8-bit DAC connected via the SPI (e.g., Texas Instrument TLC5628).

Space does not allow an in depth look at each configuration. A detailed DAC example is provided in Barrett and Sundberg [10]. In this example an external DAC0808 with supporting op amp circuitry is used to generate a ramp signal for a biomedical application. In Chapter 5 a DAC0808 is used in an industrial encoder application.

3.11 SUMMARY

In this chapter, we presented the differences between analog and digital signals and used this knowledge to discuss three sub-processing steps involved in ADC: sampling, quantization, and encoding. We also presented the quantization errors and the data rate associated with the ADC process. The dynamic range of an ADC, one of the measures to describe a conversion process, was also presented. We then presented the successive-approximation converter. Transducer interface design concepts were then discussed along with supporting information on operational amplifier configurations. We then reviewed the operation, registers, and actions required to program the ADC system aboard the ATmega328. We concluded the chapter with a discussion of the DAC process.

3.12 REFERENCES

[1] S. F. Barrett and D. J. Pack. *Atmel® AVR Microcontroller Primer Programming and Interfacing*, Synthesis Lectures on Digital Circuits and Systems, Morgan & Claypool Publishers, 2007. DOI: 10.2200/s00100ed1v01y200712dcs015. 54

[2] *Microchip ATmega328 PB AVR Microcontroller with Core Independent Peripherals and Pico Power Technology DS40001906C*, Microchip Technology Incorporation, 2018. www.microchip.com 65

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

[4] M. A. Hollander, Ed., *Electrical Signals and Systems*, 4th ed., Department of Electrical Engineering, United States Air Force Academy, McGraw-Hill Companies, Inc, 1999.

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

[6] L. Faulkenberry. *An Introduction to Operational Amplifiers*, John Wiley & Sons, New York, 1977.

[7] L. Faulkenberry. *Introduction to Operational Amplifiers with Linear Integrated Circuit Applications*, 1982. 60, 62, 63

[8] D. Stout and M. Kaufman. *Handbook of Operational Amplifier Circuit Design*, McGraw-Hill Book Company, 1976.

[9] *TLC5628C, TLC5628I Octal 8-bit Digital-to-Analog Converters*, Texas Instruments, Dallas, TX, 1997.

[10] S. F. Barrett and C. W. Sundberg. Visual feedback array to achieve reproducible limb displacements and velocities in humans, *Biomedical Sciences Instrumentation*, 53:100–105, March–April 2017. 82, 85

3.13 CHAPTER PROBLEMS

1. Given a sinusoid with 500 Hz frequency, what should be the minimum sampling frequency for an analog-to-digital converter, 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 analog-to-digital converter is 10V?

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

4. Investigate the analog-to-digital converters in your audio system. Find the sampling rate, quantization bits, and technique used for the conversion.

5. A flex sensor provides 10 K ohm of resistance for 0° flexure and 40 K ohm 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 ATmega328 ADC system.

6. If an analog signal is converted by an analog-to-digital converter 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.

7. Derive each of the characteristic equations for the classic operation amplifier configurations provided in Figure 3.4.

8. If a resistor was connected between the non-inverting terminal and ground in the inverting amplifier configuration of Figure 3.4a, how would the characteristic equation change?

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

10. 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? Explain.

11. Is an analog-to-digital converter with a better (smaller value of resolution) generally a better choice than others with a poorer resolution?

12. Research the different types of external digital-to-analog converters available for the ATmega328. Prepare a summary table.

13. Describe in detail how the ATmega328 may be quipped with a multi-channel external digital-to-analog converter.

14. A sinusoidal signal is generated with an external digital-to-analog converter. The signal's peak-to-peak value is 3 V with an average value of 1.5 V. Design a circuit to convert the signal to 5 V peak-to-peak with an average value of 0 V.

15. Repeat the question above but interchange the values of the desired input and output signals.

CHAPTER 4

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 microcontroller timer system;
- describe the procedure to capture incoming signal events;
- describe the procedure to generate time critical output signals;
- describe the timing related features of the Microchip ATmega328;
- describe the four operating modes of the Microchip ATmega328 timing system;
- describe the register configurations for the ATmega328's Timer 0, Timer 1, and Timer 2;
- program the Arduino UNO R3 using the built-in timer features of the Arduino Development Environment; and
- program the ATmega328 timer system for a variety of applications using C.

4.1 OVERVIEW

One of the most important reasons for using microcontrollers is their capability to perform time-related tasks. In a simple application, one can program a microcontroller to turn on or turn off an external device at a specific time. 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 review the capabilities of the Microchip ATmega328 microcontroller to perform time-related functions.[1] We begin with a review of timing related terminology. We then provide an overview of the general operation of a timing system followed by the timing system features aboard the ATmega328. Next, we present a detailed discussion of each of its timing channels and their different modes of operation. We then review the built-in timing functions of the ADE and conclude the chapter with a variety of examples.

[1]The sections on timing theory were adapted with permission from *Microcontroller Fundamentals for Engineers and Scientists*, S. F. Barrett and D. J. Pack, Morgan & Claypool Publishers, 2006.

4.2 TIMING-RELATED TERMINOLOGY

In this section, we review timing related terminology including frequency, period, and duty cycle.

4.2.1 FREQUENCY

Consider signal $x(t)$ that repeats itself. We call this signal periodic with period T, if it satisfies the following equation:

$$x(t) = x(t + T).$$

To measure the frequency of a periodic signal, we count the number of times a particular event repeats within a 1-s period. The unit of frequency is the Hertz or cycles per second. For example, a sinusoidal signal with a 60 Hz frequency means that a full cycle of a sinusoid signal repeats itself 60 times each second or every 16.67 ms.

4.2.2 PERIOD

The reciprocal of 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 the signal period (T), given the signal's frequency (f), we simply need to apply their inverse relationship $f = \frac{1}{T}$. Both the period and frequency of a signal are often used to specify timing constraints of microcontroller-based systems. For example, when your car is on a wintery road and slipping, the engineers who designed your car configured the anti-slippage unit to react within some millisecond period, say 20 ms. The constraint then requires the monitoring system to check slippage at a rate of 50 Hz.

4.2.3 DUTY CYCLE

In many 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 signal with a changing duty cycle over time is used. The periodic pulse signal shown in Figure 4.1a is on for 50% of the signal period and off 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 of the period a signal is on or logic high. Therefore, we refer to the signal in Figure 4.1a as a periodic pulse signal with a 50% duty cycle and the corresponding signal in (b), a periodic pulse signal with a 25% duty cycle.

4.3 TIMING SYSTEM OVERVIEW

The heart of the timing system is the time base. The time base's frequency is used to generate a baseline clock signal. For a timer system, the system clock is used to update the contents of a register called the free-running counter. The job of the free-running counter is to count up

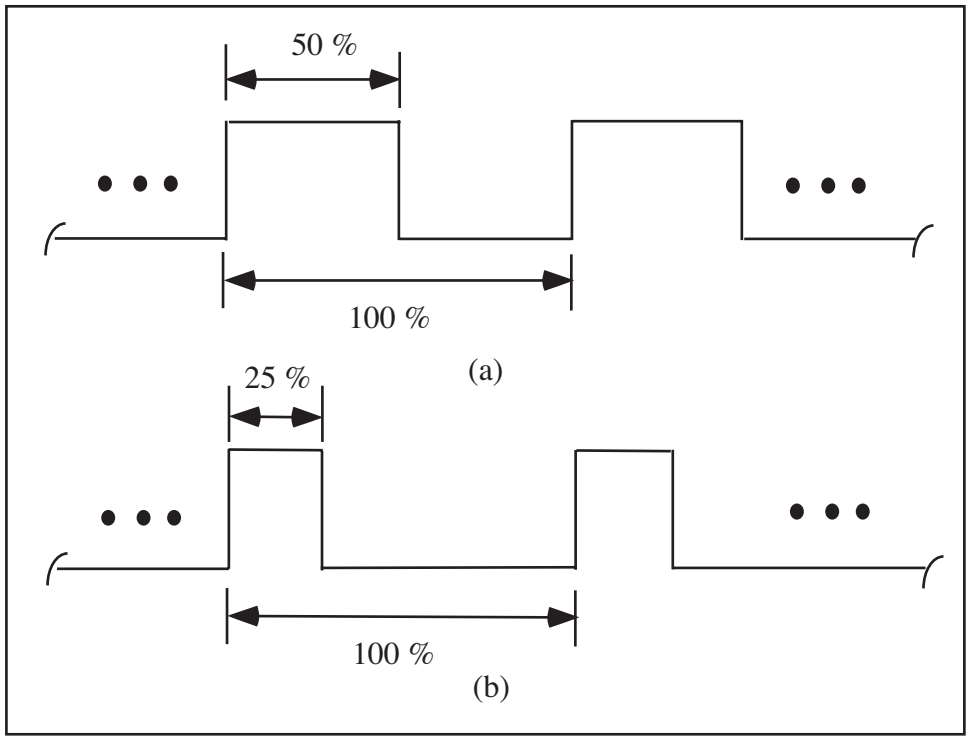


Figure 4.1: Two signals with the same period but different duty cycles. The top figure (a) shows a periodic signal with a 50% duty cycle and the lower figure (b) displays a periodic signal with a 25% duty cycle.

(increment) for each rising edge (or a falling edge) of a clock signal. For example, if a clock is operating at a rate of 2 MHz, the free-running counter will count up or increment each 0.5 ms. All other timer related events 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 ATmega328 may be clocked internally using a user-selectable RC time base or they may be clocked externally. The RC internal time base is selected using programmable fuse bits. You may choose an internal fixed clock operating frequency of 1, 2, 4, or 8 MHz. The frequency may be further divided using internal dividers.

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 an external RC network, a ceramic resonator, and a crystal oscillator. The system designer chooses the time base frequency and clock source device appropriate for the application at hand. The maximum op-

erating frequency of the ATmega328P with a 5 VDC supply voltage is 20 MHz. The Arduino UNO R3 processing board is equipped with a 16 MHz crystal oscillator time base.

The timing system may be used to capture the characteristics of an incoming input signal or generate a digital output signal. We provide a brief overview of these capabilities followed by a more detailed treatment in the next section.

For **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 the free-running counter to capture external event times. We can use these features to measure the period of an incoming signal or the width of an incoming pulse. 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 these values to easily compute the pulse width of 0.1 s.

For **output** timer functions, a microcontroller uses a comparator, a free-running counter, logic switches, 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 programmer stores a specified time in terms of the free-running counter value. The checking process is executed at each clock cycle and when a match occurs, the corresponding hardware system induces a programmed logic change on a programmed output port pin. Using such capability, one can generate a simple logic change at a designated time incident, a pulse with a desired time width, or a pulse width modulated signal to control servo or Direct Current (DC) motors.

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. The simple answer is that we cannot do so directly. A programmer must use the system clock values and derive the absolute time values. 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 ms 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. For example, 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,[2] 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, 276 and 16. 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 increment by one, we can readily compute the pulse width as 260×0.5 ms $=$ 130 ms.

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 status flag is set to notify the processor of this event. The rollover events may be counted to correctly determine the overall elapsed time of an event.

[2]The \$ symbol represents that the following value is in a hexadecimal form.

To calculate the total elapsed time of an event; the event start time, stop time, and the number of timer overflows (n) that occurred between the start time and stop time must be known. Elapsed time may be calculated using:

$$\text{elapsed clock ticks} = (n \times 2^b) + (\text{stop count} - \text{start count}) \text{ [clock ticks]}$$

$$\text{elapsed time} = (\text{elapsed clock ticks}) \times (\text{FRC clock period}) \text{ [seconds]}.$$

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.

4.4 TIMER SYSTEM APPLICATIONS

In this section, we take a closer look at some important uses of the timer system of a microcontroller to: (1) measure an input signal timing event, termed input capture; (2) to count the number of external signal occurrences; (3) to generate timed signals—termed 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.

4.4.1 INPUT CAPTURE – MEASURING EXTERNAL TIMING EVENT

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 4.2 shows an incoming periodic signal to our microcontroller.

The first step for 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 to which the incoming signal arrives. This step is done using a special input timer port configuration register. The next step is to program the input event to capture. In our current example, we should capture two consecutive rising edges or falling edges of the incoming signal. Again, the programming portion is done by storing an appropriate setup value to a special register.

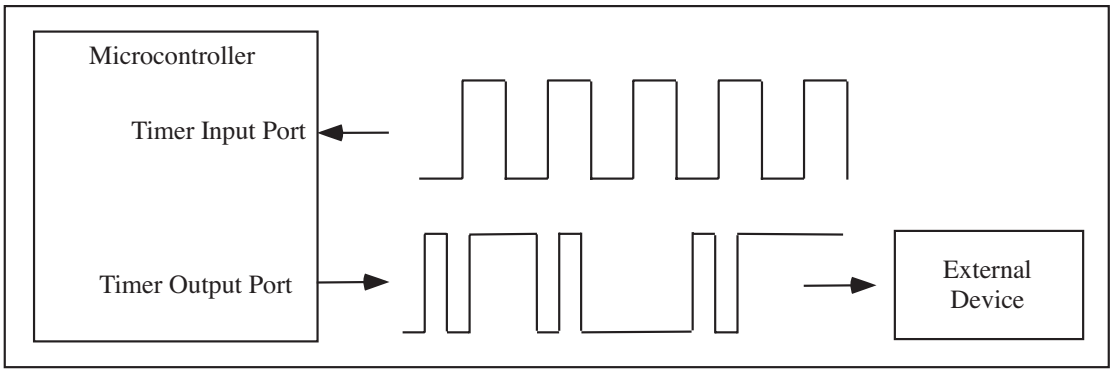


Figure 4.2: Use of the timer input and output systems of a microcontroller. The signal on top is fed into a timer input port. The captured signal is subsequently used to compute the input signal frequency. The signal on the bottom is generated using the timer output system. The signal is used to control an external device.

Now that the input timer system is configured appropriately, you have two options to accomplish the task. The first one is the use of a polling technique; 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, it needs to clear the flag and record the time when the flag was set using another special register that captures the time of the associated free-running counter value. The program needs to continue to wait for the next flag, which indicates the end of one period of the incoming signal. A 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's timer system, 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 one event. Such situation introduces the second method: interrupt systems. Most microcontrollers are equipped with built-in interrupt systems with their timer input modules. Instead of continuously polling for a flag, a microcontroller performs 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 pay the price by appropriately configuring the interrupt systems to be triggered when a desired event is detected. Typically, additional registers must be configured, and a special program called an interrupt service routine must be written.

Suppose that for an input capture scenario the two captured times for the two rising edges are $1,000 and $5,000, respectively. Note that these values are not absolute times but the time hacks captured from the free-running counter. The period of the signal is $4,000 or 16,384 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.

4.4.2 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 technique or the interrupt technique to accomplish the task. For both techniques, the initial steps of turning on the 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.

4.4.3 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 in this application, we want to send a signal shown in Figure 4.2 to turn on an external device. The timing signal 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 the 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 special 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 4.2, 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. As was mentioned in Section 7.2, 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. For the particular example in Figure 4.2, let's assume that the main clock

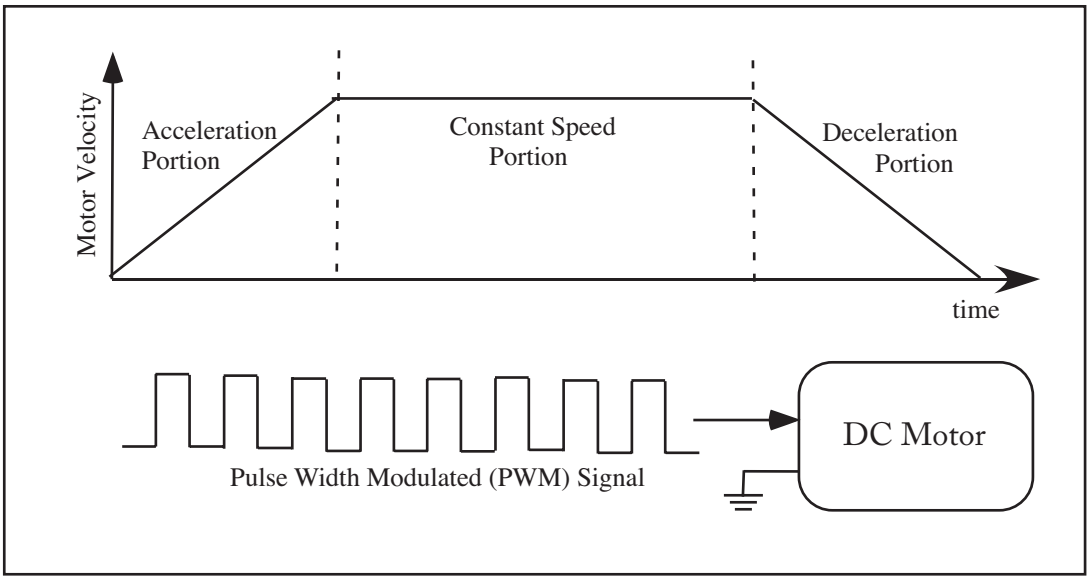


Figure 4.3: The figure shows the speed profile of a DC motor over time when a pulse-width-modulated signal is applied to the motor.

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 put $0000 first 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.

4.4.4 INDUSTRIAL IMPLEMENTATION CASE STUDY (PWM)

In this section, we discuss a well-known method to control the speed of a DC motor using a PWM signal. The underlying concept is as follows. If we turn on a DC motor and provide the required voltage, the motor will run at its maximum speed. 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. 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 4.3. As shown in the figure, we want to accelerate the speed, maintain the speed, and decelerate the speed for a fixed amount of time.

As an example, an elevator control system does not immediately operate the elevator motor at full speed. The elevator motor speed will ramp up gradually from stop to desired speed. As the elevator approaches, the desired floor it will gradually ramp back down to stop.

The first task necessary 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 want to start off with 25% duty cycle, we need to input $4,000 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 to $FFFF every 32.768 ms, and the free-running counter will count from $0000 to $FFFF approximately 30.51 times over the period of 1 s. That is we need to increase the pulse width from $4,000 to $8,000 in approximately 30 turns, or approximately 546 clock counts every turn. This technique may be used to generate any desired duty cycle.

4.5 OVERVIEW OF THE MICROCHIP ATMEGA328 TIMER SYSTEM

The Microchip ATmega328 is equipped with a flexible and powerful multiple channel timing system. For the ATmega328, the timer channels are designated Timer 0, Timer 1, and Timer 2. In this section, we review the operation of the timing system in detail. We begin with an overview of the timing system features followed by a detailed discussion of timer channel 0. Space does not permit a complete discussion of the other two types of timing channels; 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 channels.

The features of the timing system are summarized in Figure 4.4. Timer 0 and 2 are 8-bit timers, whereas Timer 1 for the ATmega 328 is a 16-bit timers. Each timing 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}).

Each timing 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

ATmega328:Timer 0	ATmega328: Timer 1	ATmega328: Timer 2
Features: - 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	**Features:** - 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	**Features:** - 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 4.4: Microchip timer system overview [www.microchip.com].

using the output compare channels. Additionally, Timer 1 on the ATmega328 is equipped with the Input Capture feature.

All of the timing 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.

4.6 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 Timers 1, 3, 4, 5, and Timer 2.

A Timer 0 block diagram is shown in Figure 4.5. The clock source for Timer 0 is provided via an external clock source at the T0 pin 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 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]

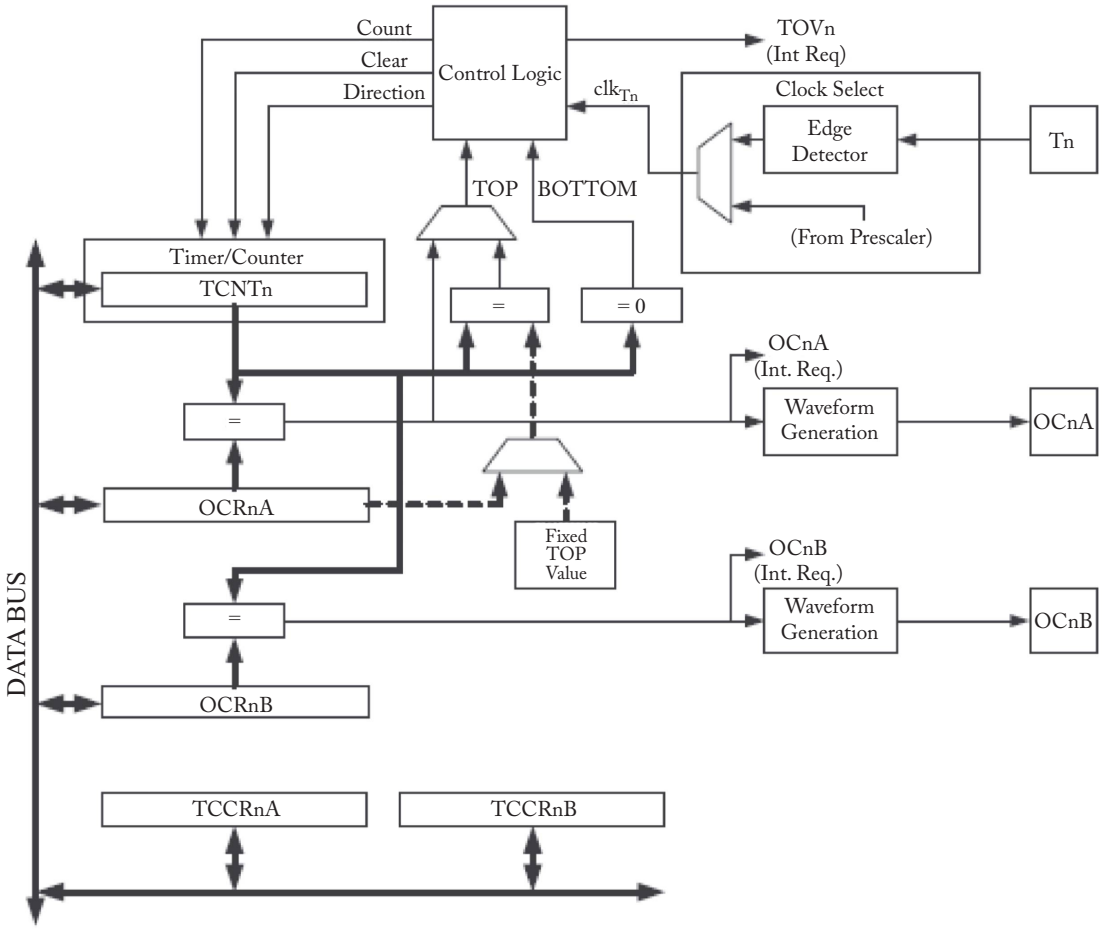


Figure 4.5: Timer 0 block diagram. (Figure used with permission Microchip, Inc. [www.microchip.com].)

(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 4.5, 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 signal interrupt related events.

4.6.1 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 4.6. 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).

4.6.1.1 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 provide delays within a system. We provide an example of this application in Section 5.9.

4.6.1.2 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.

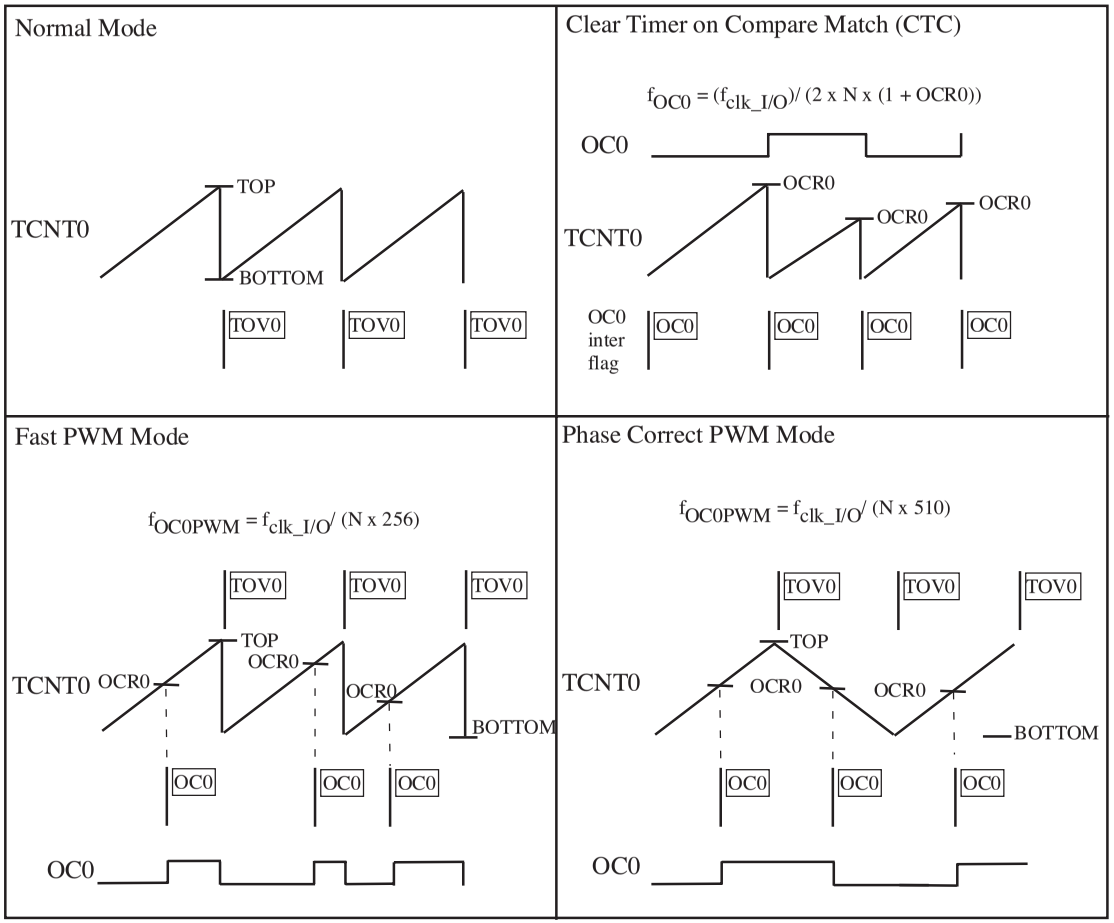


Figure 4.6: Timer 0 modes of operation [www.microchip.com].

4.6.1.3 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.

4.6.1.4 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.

4.6.2 TIMER 0 REGISTERS

A summary of the Timer 0 registers are shown in Figure 4.7.

4.6.2.1 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 4.8.

4.6.2.2 Timer/Counter Register (TCNT0)

The TCNT0 is the 8-bit counter for Timer 0.

4.6.2.3 Output Compare Registers A and B (OCR0A and OCR0B)

The OCR0A and B registers holds a user-defined 8-bit value that is continuously compared to the TCNT0 register.

4.6.2.4 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.

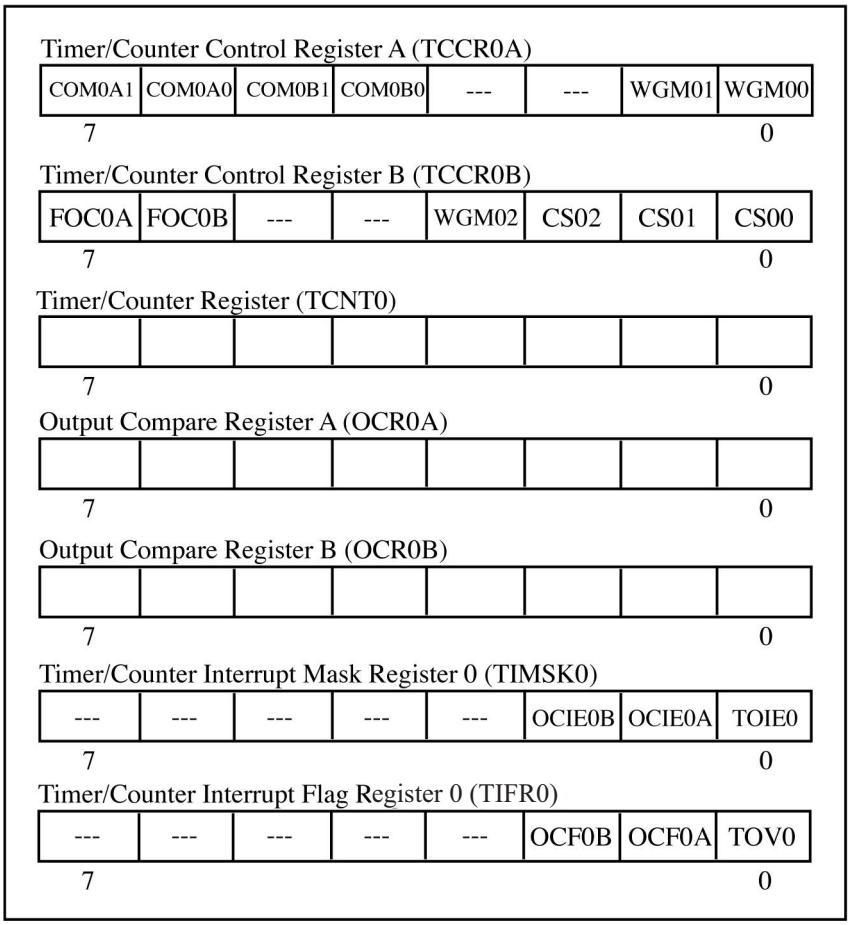


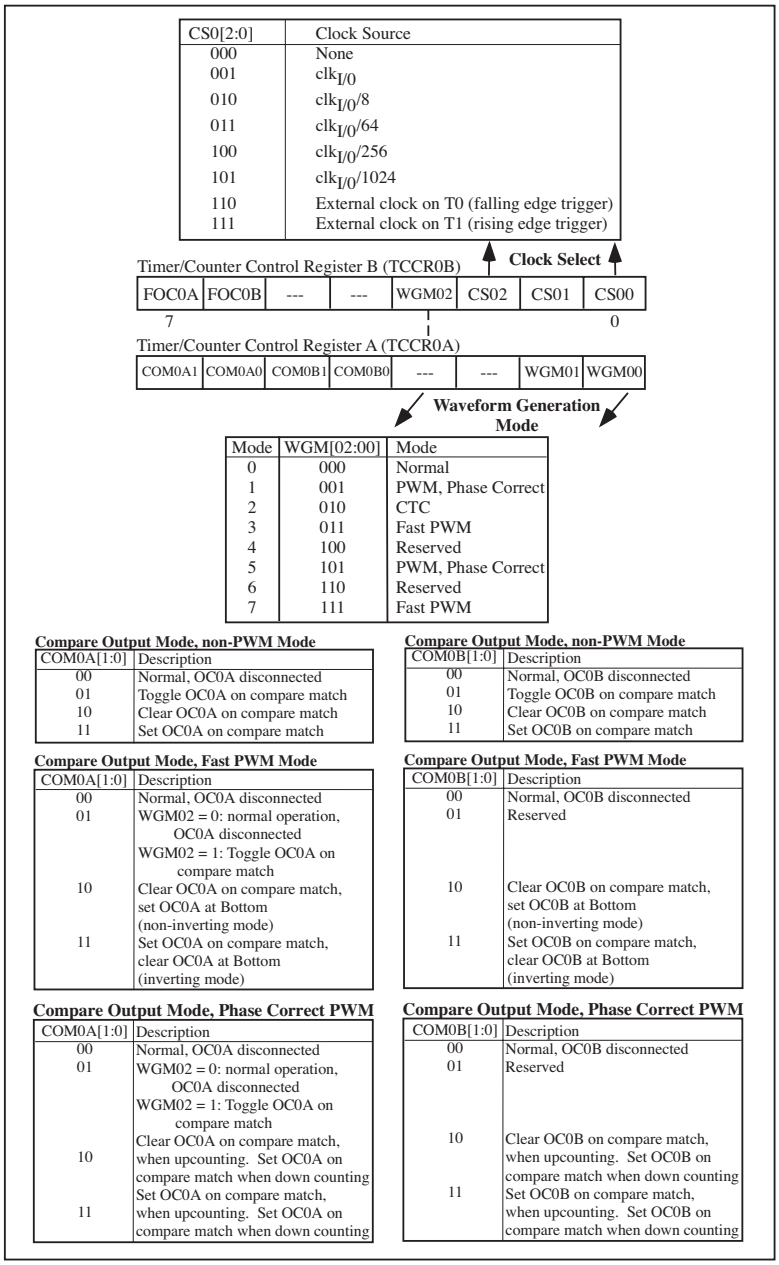
Figure 4.7: Timer 0 registers [www.microchip.com].

4.6.2.5 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 Timer/Counter 0 overflows.

4.7 TIMER 1

Timer 1 on the ATmega328 is a 16-bit timer/counters. These timers share 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



CS0[2:0]	Clock Source
000	None
001	$clk_{I/O}$
010	$clk_{I/O}/8$
011	$clk_{I/O}/64$
100	$clk_{I/O}/256$
101	$clk_{I/O}/1024$
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
7							0

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 4.8: Timer/Counter Control Registers A and B (TCCR0A and TCCR0B) bit settings [www.microchip.com].

output compare channel and also the capability for input capture. The block diagram for Timer 1 is shown in Figure 4.9.

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).

4.7.1 TIMER 1 REGISTERS

The complement of registers supporting Timer 1 are shown in Figure 4.10. Each register will be discussed in turn.

4.7.1.1 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;
- 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 4.11.

4.7.1.2 Timer/Counter Register 1 (TCNT1H/TCNT1L)

The TCNT1 is the 16-bit counter for Timer 1.

4.7.1.3 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.

4.7.1.4 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.

4.7.1.5 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.

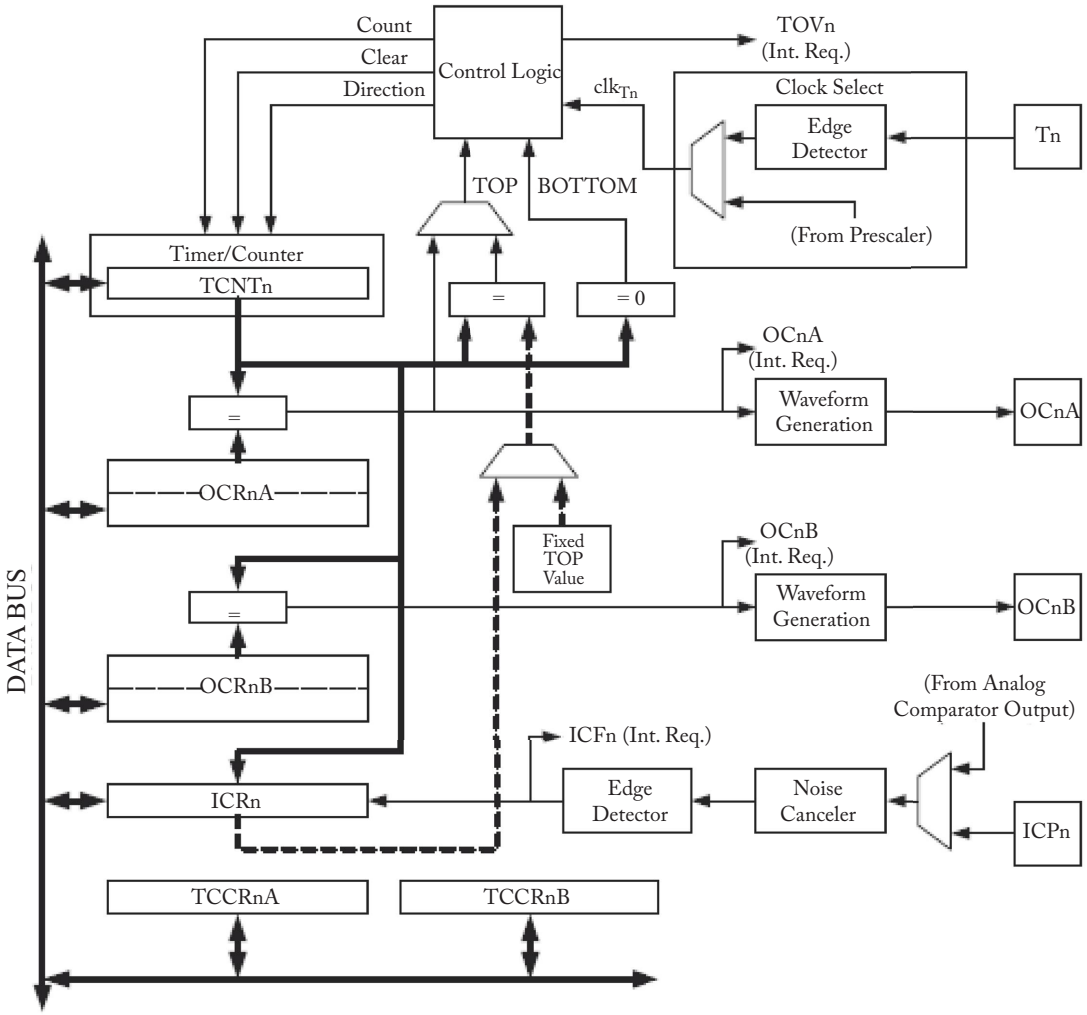


Figure 4.9: Timer 1 block diagram. (Figure used with Permission, Microchip, Inc. [www.microchip.com].)

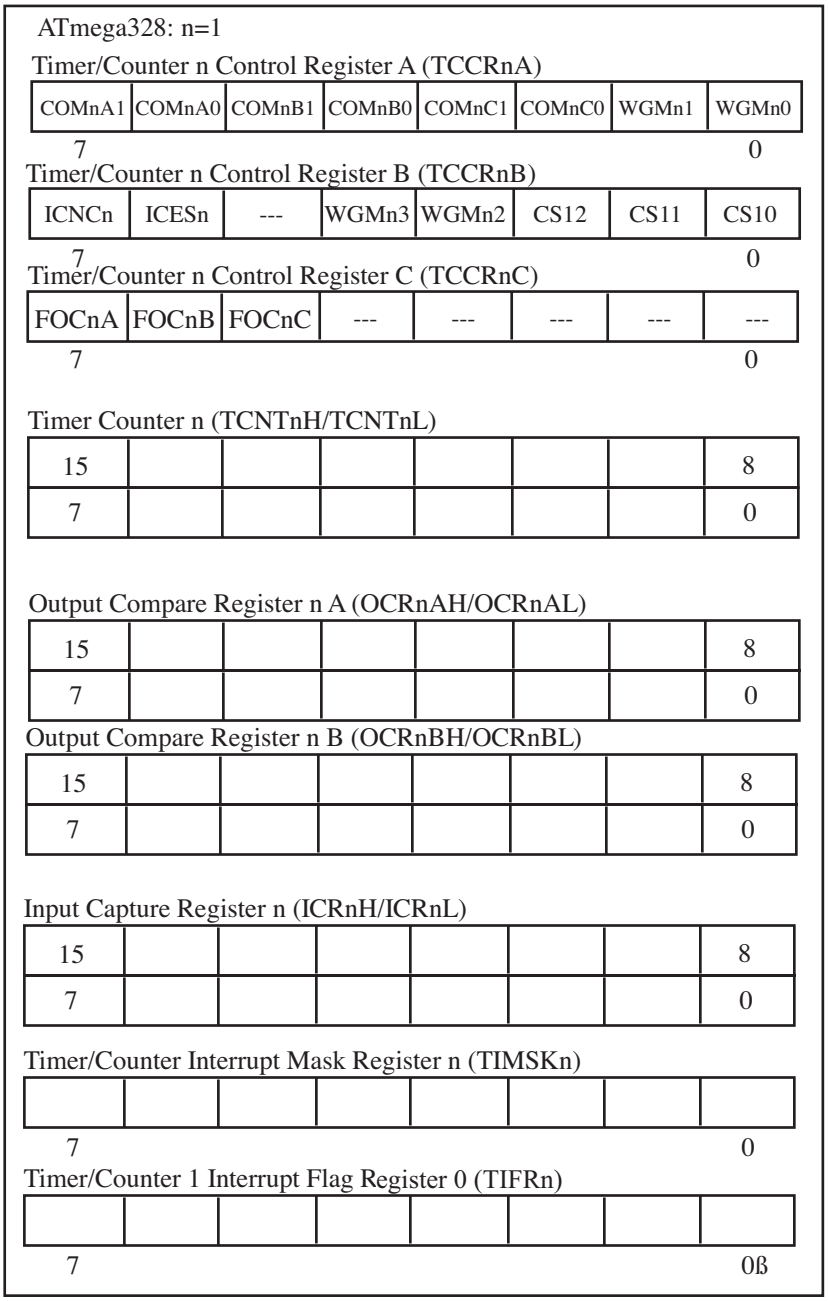
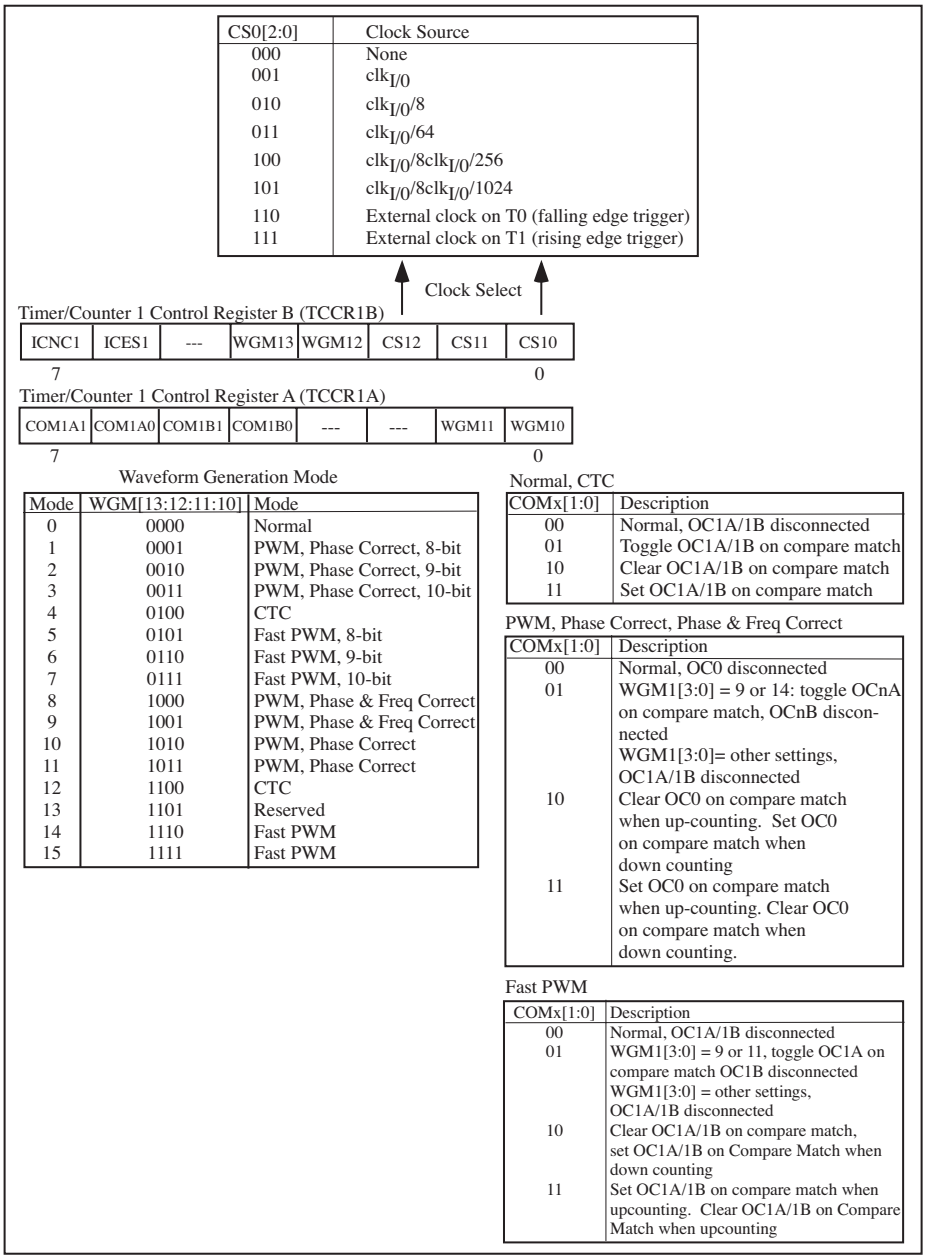


Figure 4.10: Timer 1 registers [www.microchip.com].



CS0[2:0]	Clock Source
000	None
001	$clk_{I/O}$
010	$clk_{I/O}/8$
011	$clk_{I/O}/64$
100	$clk_{I/O}/8 clk_{I/O}/256$
101	$clk_{I/O}/8 clk_{I/O}/1024$
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	---	---	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] = 9 or 11, 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 on Compare Match when down counting
11	Set OC1A/1B on compare match when upcounting. Clear OC1A/1B on Compare Match when upcounting

Figure 4.11: TCCR1A and TCCR1B registers [www.microchip.com].

4.7.1.6 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.

4.7.1.7 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.

4.8 TIMER 2

Timer 2 is another 8-bit timer channel similar to Timer 0. The Timer 2 channel block diagram is provided in Figure 4.12. Its registers are summarized in Figure 4.13.

4.8.0.1 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;
- 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 4.14.

4.8.0.2 Timer/Counter Register(TCNT2)

The TCNT2 is the 8-bit counter for Timer 2.

4.8.0.3 Output Compare Register A and B (OCR2A and B)

The OCR2A and B registers hold a user-defined 8-bit value that is continuously compared to the TCNT2 register.

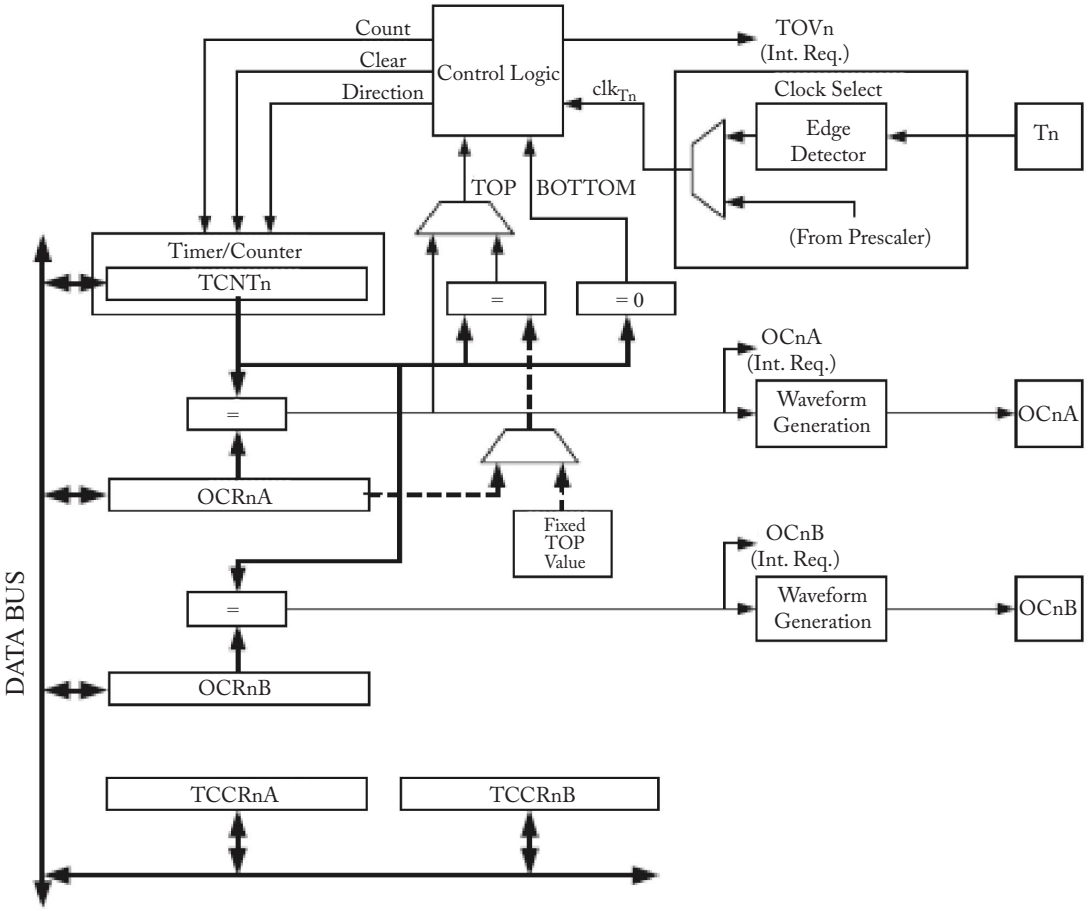


Figure 4.12: Timer 2 block diagram. (Figure used with Permission, Microchip, Inc. [www.microchip.com].)

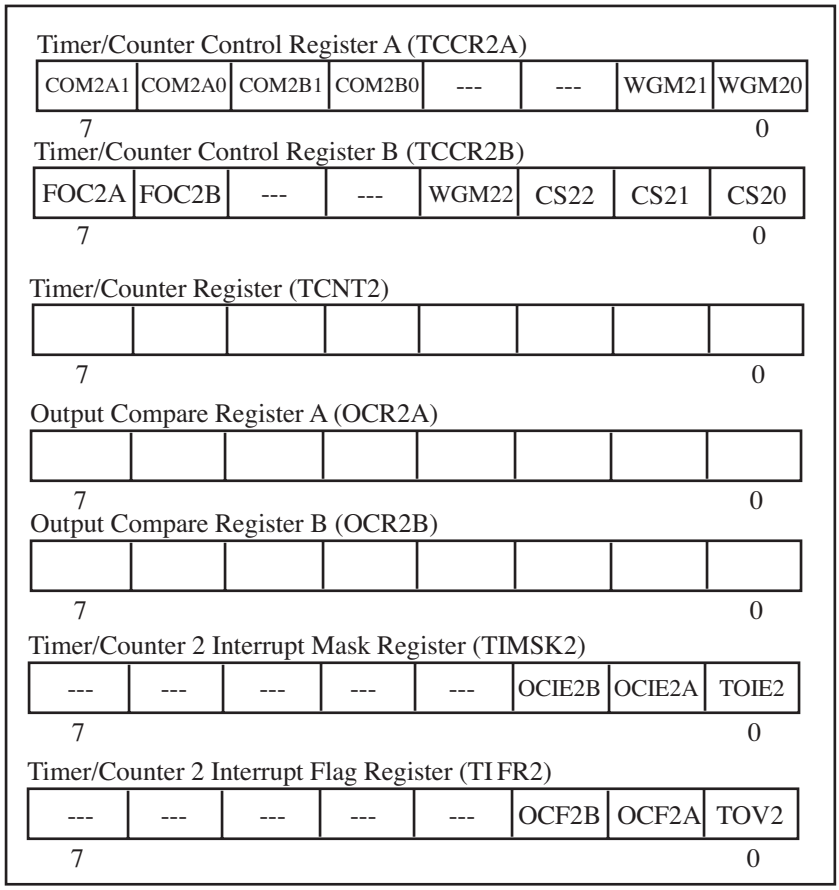


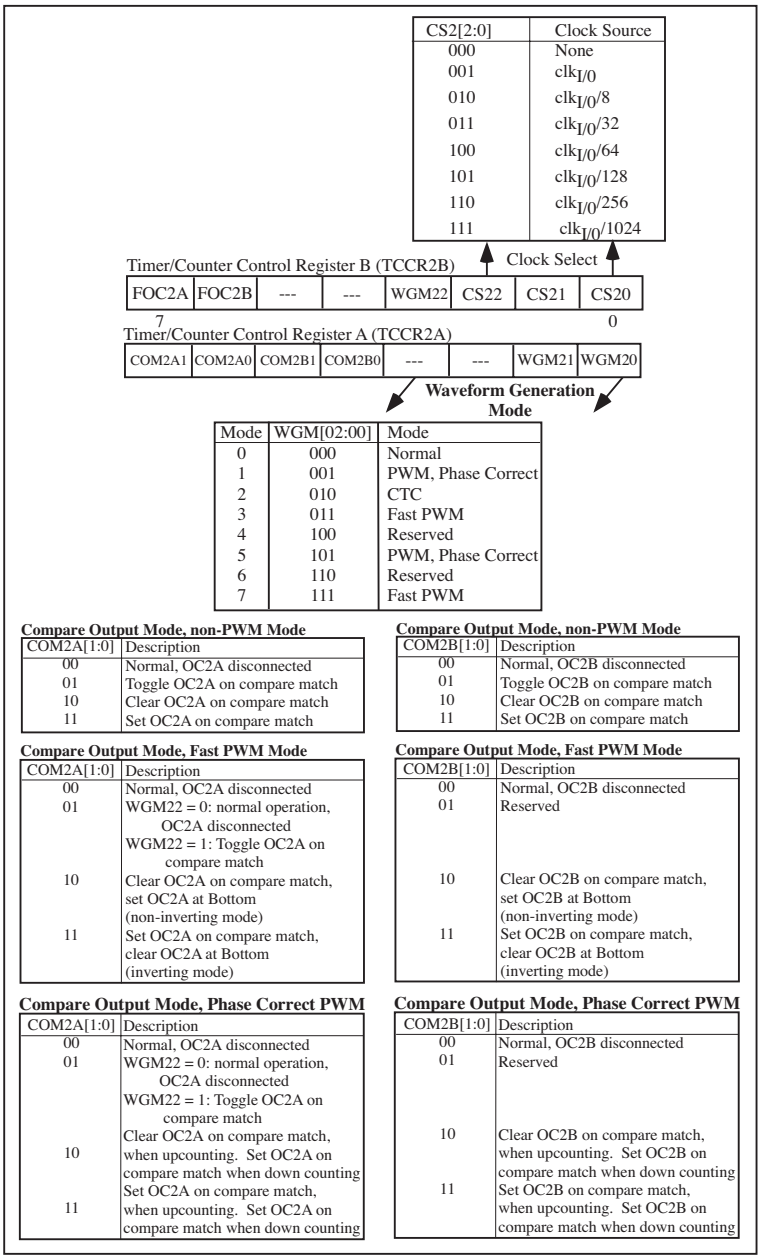
Figure 4.13: Timer 2 registers [www.microchip.com].

4.8.0.4 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.

4.8.0.5 Timer/Counter Interrupt Flag Register 2 (TIFR2)

Timer 2 uses the Output Compare Flags 2 A and B (OCF2A and B) which sets for an output compare match. Timer 2 also uses the Timer/Counter 2 Overflow Flag (TOV2) which sets when Timer/Counter 2 overflows.



CS2[2:0]	Clock Source
000	None
001	$clk_{I/O}$
010	$clk_{I/O}/8$
011	$clk_{I/O}/32$
100	$clk_{I/O}/64$
101	$clk_{I/O}/128$
110	$clk_{I/O}/256$
111	$clk_{I/O}/1024$

Clock Select

Timer/Counter Control Register B (TCCR2B)

FOC2A	FOC2B	---	---	WGM22	CS22	CS21	CS20
7							0

Timer/Counter Control Register A (TCCR2A)

COM2A1	COM2A0	COM2B1	COM2B0	---	---	WGM21	WGM20

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

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 4.14: Timer/Counter Control Register A and B (TCCR2A and B) bit settings [www.microchip.com].

4.9 PROGRAMMING THE ARDUINO UNO R3 USING THE BUILT-IN ARDUINO DEVELOPMENT ENVIRONMENT TIMING FEATURES

The ADE has several built-in timing features. These include the following.

- **delay(unsigned long):** The delay function pauses a sketch for the amount of time specified in milliseconds.
- **delayMicroseconds(unsigned int):** The delayMicroseconds function pauses a sketch for the amount of time specified in microseconds.
- **pulseIn(pin, value):** The pulseIn function measures the length of an incoming digital pulse. If value is specified as HIGH, the function waits for the specified pin to go high and then times until the pin goes low. The pulseIn function returns the length of elapsed time in microseconds as an unsigned long.
- **analogWrite(pin, value):** The analog write function provides a pulse width modulated (PWM) output signal on the specified pin. The PWM frequency is approximately 490 Hz. The duty cycle is specified from 0 (value of 0) to 100 (value of 255) %.

Example: The Arduino IDE contains an example to control the intensity of an LED using the **analogWrite** function. Go to: File–>Examples–>01.Basics–>Fade.

4.10 PROGRAMMING THE TIMER SYSTEM IN C

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.[3]

4.10.1 PRECISION DELAY

A precision delay may be generated by counting the number of elapsed interrupts and Timer overflows. In the following example, the ATmega328 is configured to interrupt 65.5 ms. Different length delays are then created by pausing a specified number of interrupts. For example, to delay for a second requires approximately fifteen 65.5 ms interrupts. The interrupt period will change when the microcontroller is clocked at a different frequency. This code example may be adjusted to different clock frequencies and different desired interrupt intervals.

The following steps are followed to achieve a desired time delay.

[3]Examples were originally developed for the ATmega164 and provided in *Microchip AVR Microcontroller Primer: Programming and Interfacing*, S. F. Barrett and D. J. Pack, 3rd ed., 2019. The examples were adapted with permission for the ATmega328.

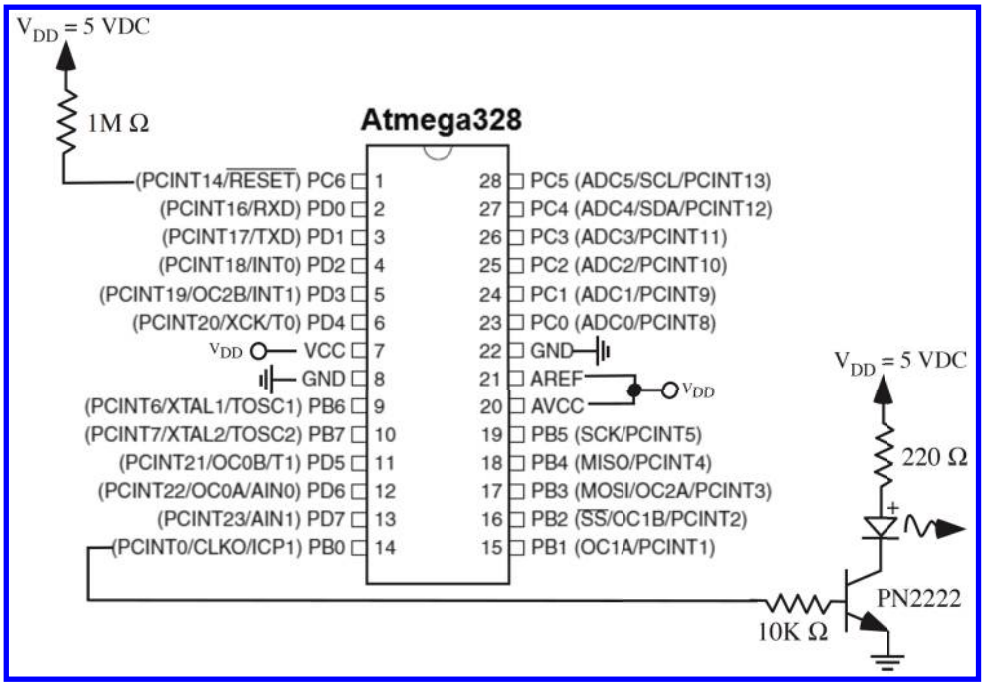


Figure 4.15: LED connected to PORTB[0].

1. Choose a desired time base value for the microcontroller. In general, the slowest frequency appropriate for a specific application should be chosen. Power consumption of CMOS based circuitry is linearly related to operating frequency. Also, choose the type of desired time base for inherent accuracy (internal oscillator, external resonator, or external crystal oscillator).

2. Choose a timer channel to generate the delay. In the following example Timer Channel 0 is used. It is an 8-bit timer with an 8-bit TCNT0 register. The TCNT0 register will overflow every 2^8 counts or 256 counts.

3. Additional time division may be set using Timer 0 registers (e.g., TCCR0B).

4. With all parameters set determine the interrupt time interval.

Example: In this example we program the ATmega328 to provide a delay of some number of 65.5 ms interrupts. The Timer 0 overflow is configured to occur every 65.5 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 PORTB[0], as shown in Figure 4.15.

To arrive at the 65.5 ms interrupt time increment, the following steps were taken.

1. The internal 8 MHz internal oscillator was chosen with the divide-by-8 option using internal fuse settings.
2. The resulting 1 MHz time base was further divided by 256 using the TCCR0B register.
3. Using TCNT0, the timer overflow interrupt would set every 256 counts or 65.5 ms.

```
//*************************************************************
//flash_led.c
//*************************************************************

#include <iom328pv.h>
#include <macros.h>

//function prototypes*************************************************
void delay(unsigned int number_of_65_5ms_interrupts);
void init_timer0_ovf_interrupt(void);
void timer0_interrupt_isr(void);
void delay(unsigned int number_of_65_5ms_interrupts);
void initialize_ports(void);

//initialize timer0 overflow interrupt*****************************
                                    //interrupt handler def
#pragma interrupt_handler timer0_interrupt_isr:17


//global variables*****************************************************
unsigned int   input_delay;         //counts num of Timer/Ctr 0
                                    //Overflow interrupts

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

void main(void)
{
init_timer0_ovf_interrupt();  //initialize Timer/Counter0 Overflow
                              //interrupt - call once at beginning
initialize_ports();
PORTB = PORTB | 0x01;         //PORTD[0] high

while(1)
  {
```

```
  delay(15);                    //1 second delay
  PORTB = PORTB & 0xFE;         //PORTB[0] low

  delay(15);                    //1 second delay
  PORTB = PORTB | 0x01;         //PORTB[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 internal 8 MHz RC oscillator with the divide by 8
//fuse set 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 65.5 ms
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_65_5ms_interrupts): this generic delay
//function provides the specified delay as the number of 65.5 ms
//"clock ticks" from the Timer/Counter0 Overflow interrupt.
```

```
//Note: This time delay is only accurate when the internal 8 MHz
//RC oscillator is used with the divide by 8 fuse set and
//then divided by 256.  The 8-bit Timer0 register (TCNT0)
//overflows every 256 counts or every 65.5 ms.
//*****************************************************************

void delay(unsigned int number_of_65_5ms_interrupts)
{
TCNT0 = 0x00;                    //reset timer0
input_delay = 0;                 //reset timer0 overflow counter
while(input_delay <= number_of_65_5ms_interrupts)
  {
  ;                              //wait for number of interrupts
  }
}

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

void initialize_ports(void)
{
DDRB =0xff; //set PORTB 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
}

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

4.10.2 PULSE WIDTH MODULATION

Example: In this example, PWM signals are generated on OC1A (PORTB[1]) and OC1B (PORTB[2]) pins. The ATmega328P 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 (PORTB[1]) and OC1B (PORTB[2])
// - Internal 8 MHz oscillator with divide-by-eight fuse set
// - Overall operating frequency 1 MHz
//*****************************************************************

#include <iom328pv.h>

//function prototypes************************************************
void PWM_test(void);
void initialize_ports(void);

void main(void)
{
initialize_ports();
PWM_test();

while(1)
  {
  ;
  }

}

//*****************************************************************
//void PWM_test(void): the PWM is configured to generate PWM signals
//on the OC1A (PORTB[1]) and OC1B (PORTB[2]) pins.  The ATmega328P
//is set for the internal 8 MHz oscillator with divide by eight fuze
//set resulting in an overall operating frequency of 1 MHz.
//*****************************************************************

void PWM_test(void)
{
TCCR1A = 0xA1;                    //freq = resonator/510=1 MHz/510
                                  //freq = 1.9607 kHz
TCCR1B = 0x03;                    //no clock source division
                                  //Init PWM duty cycle variables
                                  //Set PWM to 50
OCR1BH = 0x00;                    //PWM duty cycle CH B
```

```
OCR1BL = (unsigned char)(128);
OCR1AH = 0x00;                      //PWM duty cycle CH A
OCR1A = (unsigned char)(128);
}

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

void initialize_ports(void)
{
DDRB =0xff;  //set PORTB as output
PORTB=0x01;  //initialize low

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 (OC1B) is used to generate a pulse width modulated signal on PORTB[2] (pin 16). An analog voltage provided to ADC Channel 3 PORTC[3] (pin 26) is used to set the desired duty cycle from 50–100%, as shown in Figure 4.16. The ATmega328P is set for the internal 8 MHz oscillator with divide by eight fuze set resulting in an overall operating frequency of 1 MHz.

```
//*****************************************************************
//pwm_adc.c: Timer 1, Channel B (OC1B) is used to generate a PWM
//signal on PORTB[2] (pin 16).  An analog voltage provided to
//ADC Channel 3 PORTC[3] (pin 26) is used to set the  desired duty
//cycle from 50 to 100
// The ATmega328P is set for the internal 8 MHz oscillator with
//divide by eight fuze set resulting in an overall operating
//frequency of 1 MHz.
//*****************************************************************

#include <iom328pv.h>
```

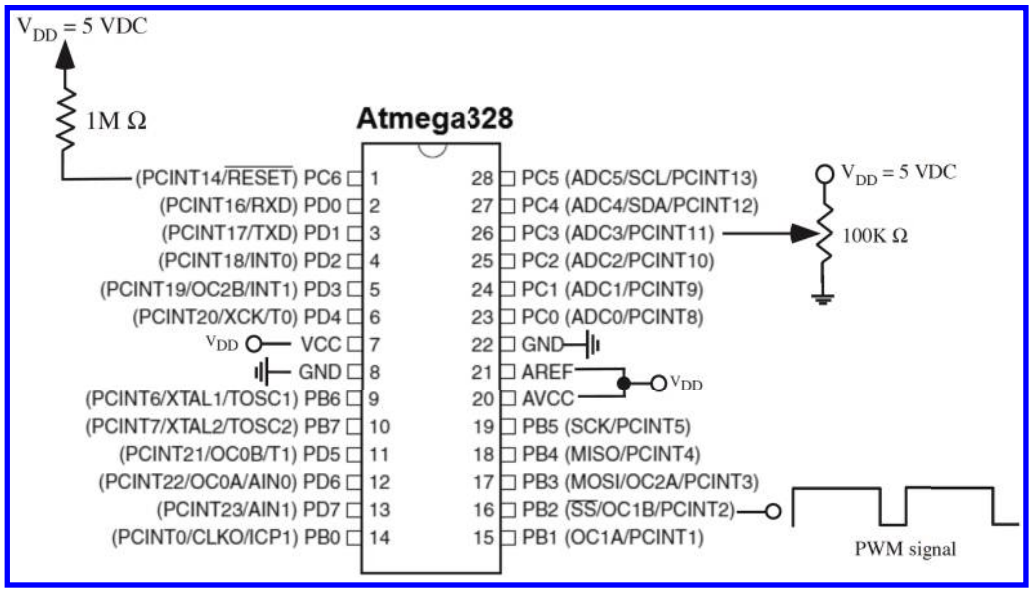


Figure 4.16: Pulse width modulation.

```
//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)
{
DDRB =0xff; //set PORTB 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 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 max duty cycle setting
                                    //0 VDC = 50
```

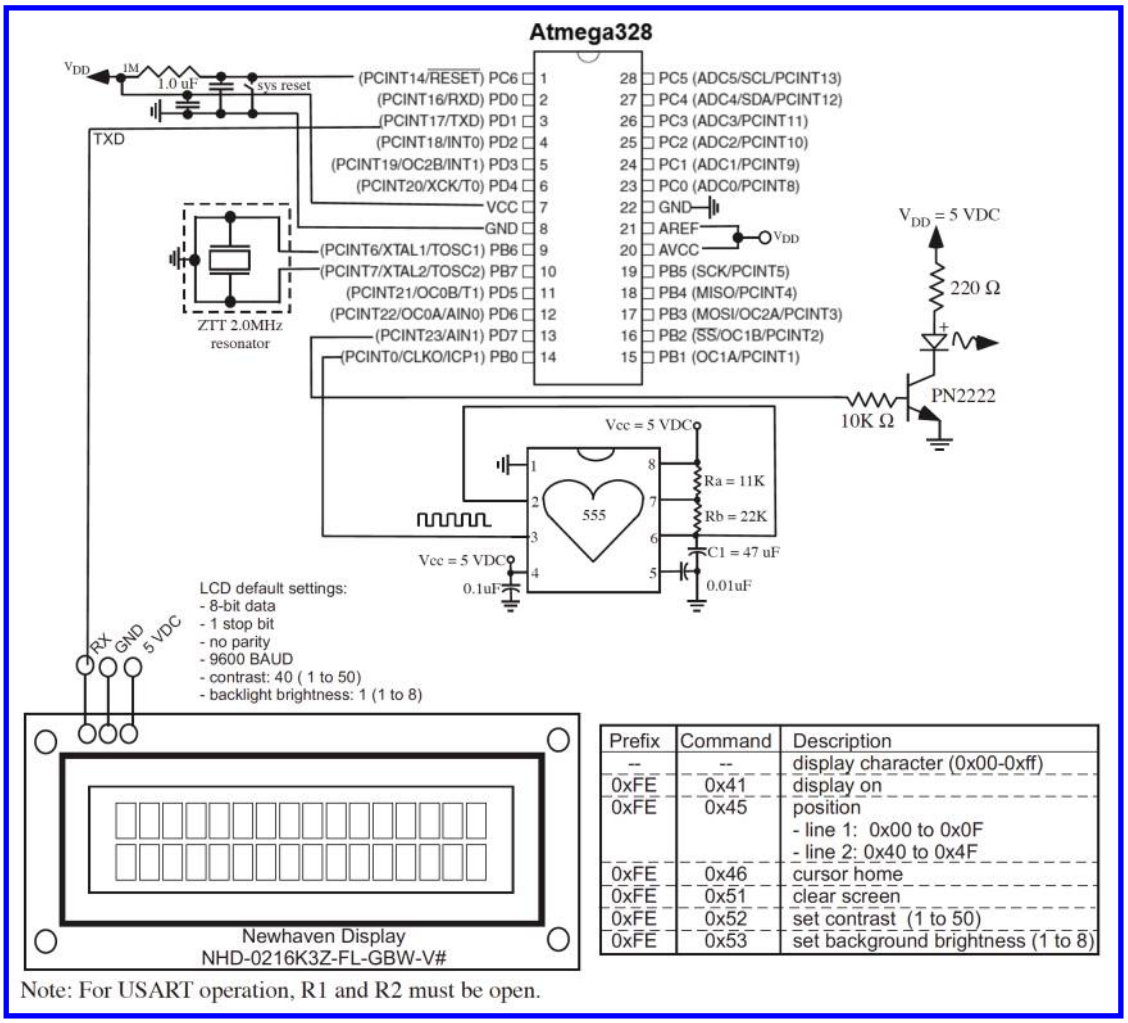
```
speed_float = ((float)(speed_int)/(float)(0x0400));
                                    //Convert to PWM constant 127-255
speed_int = (unsigned int)((speed_float * 127) + 128.0);
                                    //Configure PWM clock
TCCR1A = 0xA1;                      //freq = int osc/64 = 1 MHz/64
                                    //freq = 15.6 kHz
TCCR1B = 0x03;                      //no clock source division
                                    //Init PWM duty cycle variables
OCR1BH = 0x00;
OCR1BL = (unsigned char)(speed_int);//Set PWM duty cycle CH B to 0

}

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

4.10.3 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 (PORTB[0], ICP1) is being used to monitor the heart rate (typically 50–120 beats per minute) of a patient. The microcontroller is clocked by an external 2 MHz ceramic resonator. The 555 timer is used to simulate a heartbeat as shown in Figure 4.17. Results are displayed on a serial LCD module (Newhaven NHD-0216K3Z-FL-GBW-V3, www.newhavendisplay.com). The ATmega328 communicates with the serial LCD via Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) channel 0. USART details are provided in the next chapter.

The resulting waveforms are provided in Figure 4.18. Note how ICP1 is triggered on every rising edge of the heart beat signal from the 555 timer.

```
//***************************************************************
//file name: heartbeat3.c
//ATmega328 clocked by an external 2.0 MHz ceramic resonator
//***************************************************************

//include files*************************************************

//MICROCHIP register definitions for ATmega328
#include <iom328pv.h>

//function prototypes*******************************************
void delay(unsigned int number_of_32_8ms_interrupts);
```



Prefix	Command	Description
--	--	display character (0x00-0xff)
0xFE	0x41	display on
0xFE	0x45	position - line 1: 0x00 to 0x0F - line 2: 0x40 to 0x4F
0xFE	0x46	cursor home
0xFE	0x51	clear screen
0xFE	0x52	set contrast (1 to 50)
0xFE	0x53	set background brightness (1 to 8)

Note: For USART operation, R1 and R2 must be open.

Figure 4.17: Heartbeat simulator.

```
void init_timer0_ovf_interrupt(void);
void initialize_ports(void);
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);
```

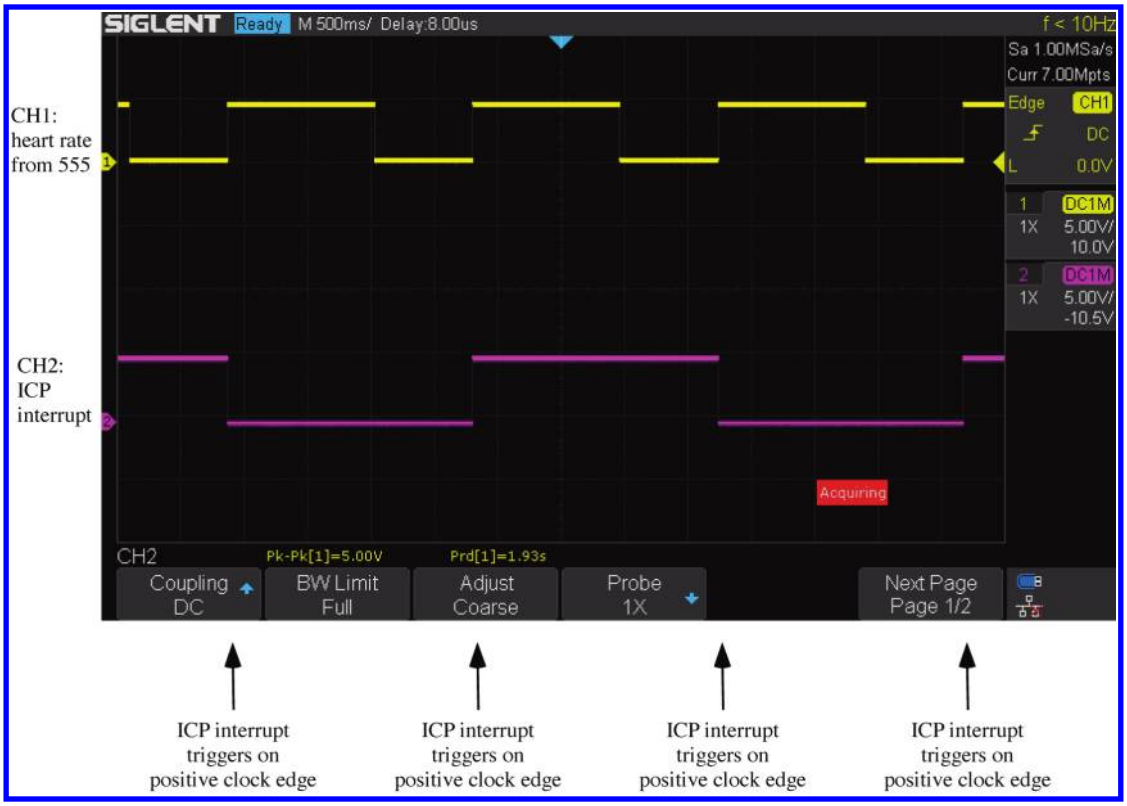


Figure 4.18: Heartbeat ICP1 waveforms. Signal captured with a Siglent SDS 1104X-E digital storage oscilloscope, four channel, 100 MHz.

```
void USART_transmit(unsigned char data);
void LCD_init(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:17
#pragma interrupt_handler input_capture_ISR:11

//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();
init_timer0_ovf_interrupt();           //init Timer0 for delay

clear_LCD();

initialize_ICP_interrupt();            //init input capture int

delay(30);                             //delay 1s
PORTD = PORTD | 0x80;                  //LED ON PORTD[7]


while(1)
  {
  delay(180);                           //delay 5s
  print_heart_rate();
  }
}

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

void initialize_ports(void)
{
```

```
DDRB =0xfe;  //PORTB[0] as input, set PORTB[7:1] 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_32_8_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 2.0 MHz
//time base.
//*****************************************************************

void delay(unsigned int number_of_32_8ms_interrupts)
{
TCNT0 = 0x00;                          //reset delay_timer
delay_timer = 0;
while(delay_timer <= number_of_32_8ms_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 2.0 MHz and
//then is divided by 256.  The 8-bit Timer0 register (TCNT0)
//overflows every 256 counts or every 32.8 ms.
//*****************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04; //divide timer0 timebase by 256, overflow occurs
```

```
                //every 32.8 ms
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)
{
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 = 2.0 MHz ceramic resonator
//       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 = 0.51 ms
    {                            //Divide by 1953 to get seconds/pulse
    HR=60/(time_pulses/1953);    //(secs/min)/(secs/beat) =bpm
    }
  else
    {
```

```
    HR=0;
    }
  }
  else
  {
  HR=0;
  }
}

//*****************************************************************
//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
}

//*****************************************************************
//USART_init: initializes the USART system
//*****************************************************************
void USART_init(void)
{
UCSR0A = 0x00;                        //control register init
UCSR0B = 0x08;                        //enable transmitter
UCSR0C = 0x06;                        //async, no parity, 1 stop bit
                                      //8 data bits
                                      //Baud Rate initialization
UBRR0H = 0x00; UBRR0L = 0x0c;         //9600 BAUD, 2 MHz clocl
                                      //set divider to 12 (0x0c)
}

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

void USART_transmit(unsigned char data)
{
while((UCSR0A & 0x20)==0x00) //wait for UDRE flag
  {
  ;
  }
UDR0 = data;                           //load data to UDR for tx
}

//*****************************************************************
//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);
}

//*****************************************************************
//void clear_LCD(void)
//*****************************************************************

void clear_LCD(void)
{
USART_transmit(0xFE);
USART_transmit(0x51);
```

```
}

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

4.11 EXAMPLE: SERVO MOTOR CONTROL WITH THE PWM SYSTEM IN C

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 4.19. 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 4.19. The PB0 and PB1 inputs of the ATmega328 provide for clockwise (CW) and counter-clockwise (CCW) rotation of the servo motor, respectively. The time base for the ATmega328 is provided by a 128 KHz external RC oscillator. Also, the external 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 ATmega328. A voltage-follower op amp circuit is used as a buffer between the ATmega328 and the servo motor.

The software to support the test and interface circuit is provided below.

```
//*************************************************************
//target controller: MICROCHIP ATmega328
//
//MICROCHIP AVR ATmega328PV Controller Pin Assignments
//Chip Port Function I/O Source/Dest Asserted Notes
//Pin  1 PUR Reset - 1M resistor to Vdd, tact switch to ground,
//       1.0 uF to ground
//Pin  7 Vdd - 1.0 uF to ground
//Pin  8 Gnd
//Pin  9 PB6 ceramic resonator connection
//Pin 10 PB7 ceramic resonator connection
//PORTB:
//Pin 14 PB0 to active high RC debounced switch - CW
//Pin 15 PB1 to active high RC debounced switch - CCW
//Pin 16 PB2 - to servo control input
//Pin 20 AVcc to Vdd
//Pin 21 ARef to Vdd
//Pin 22 AGnd to Ground
//
```

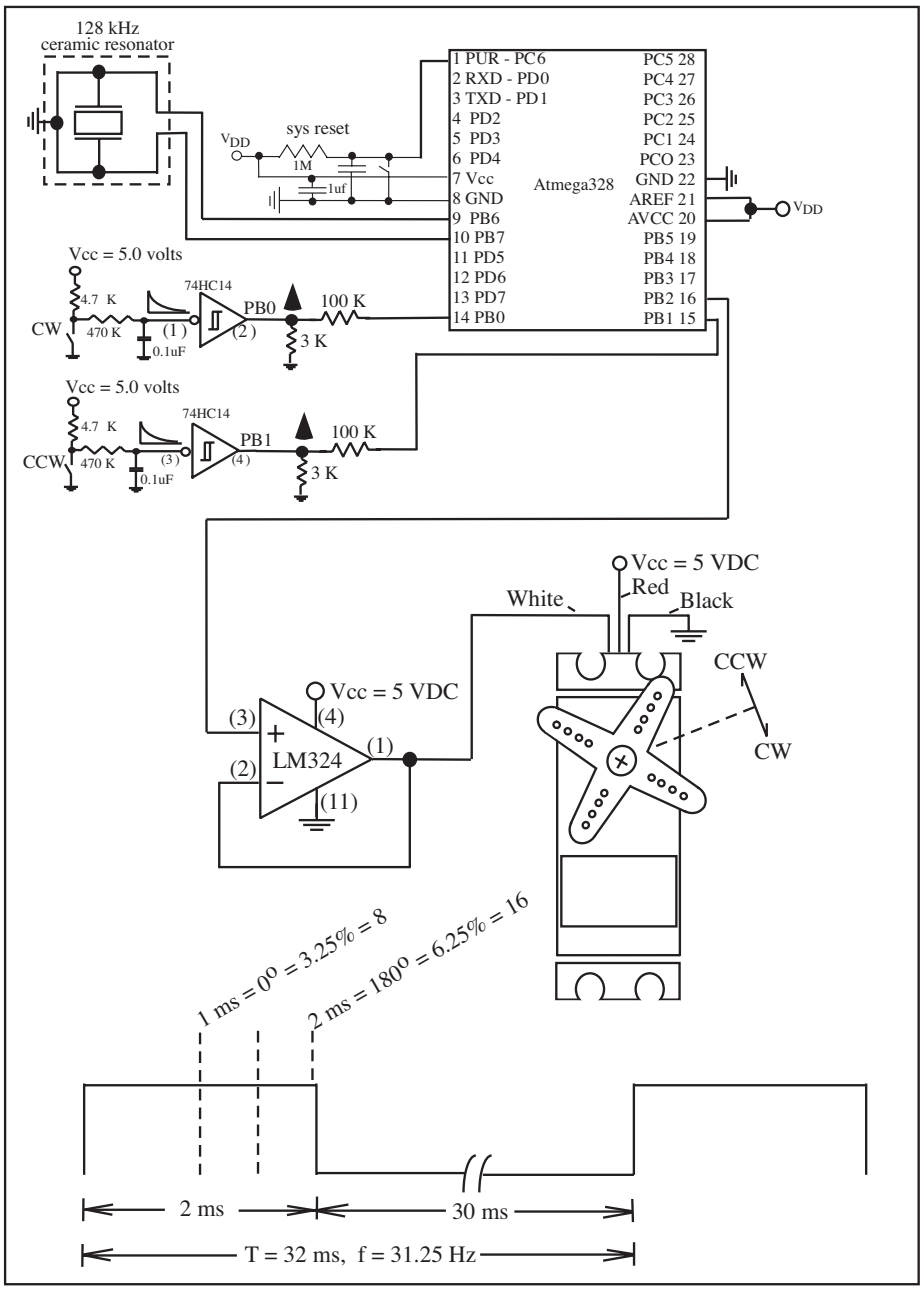


Figure 4.19: Test and interface circuit for a servo motor.

```
//include files*********************************************************
//MICROCHIP register definitions for ATmega328
#include<iom328pv.h>
#include<macros.h>

//function prototypes***************************************************
void initialize_ports(void);           //initializes ports
void read_new_input(void);             //read input change PORTB
void init_timer0_ovf_interrupt(void);  //init timer0 overflow

//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)
{
initialize_ports();                    //port config to default
                                       //external ceramic resonator:
                                       //   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
                                       //= 8 counts to
                                       //6.2
                                       //= 16 counts
                                       //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 stack overflow
  read_new_input();                     //read input status changes
  }
}//end main

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

void initialize_ports(void)
{
//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;                             //initialize 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
    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
}

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

4.12 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 ATmega328 and reviewed the features, operation, registers, and programming of the three different types of timer channels. We then investigated the built-in timing features of the ADE. We concluded with an example employing a servo motor controlled by the ATmega328 PWM system and an inexpensive laser light show.

4.13 REFERENCES

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

[2] S. F. Barrett and D. J. Pack. *Atmel® AVR Microcontroller Primer Programming and Interfacing*, Morgan & Claypool Publishers, 2008. DOI: 10.2200/s00100ed1v01y200712dcs015.

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

[4] *Microchip ATmega328 PB AVR Microcontroller with Core Independent Peripherals and Pico Power Technology DS40001906C*, Microchip Technology Incorporation, 2018. www.microchip.com

[5] S. F. Barrett and D. J. Pack. *Microchip AVR Microcontroller Primer: Programming and Interfacing*, 3rd ed., Morgan & Claypool Publishers, 2019.

4.14 CHAPTER PROBLEMS

1. What are the trade-offs of using the internal RC oscillator vs. an external time base (e.g., ceramic resonator, crystal oscillator) for the ATmega328 time base?
2. 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.
3. If we desire to generate periodic signals with periods ranging from 125 ns to 500 ms, what is the minimum frequency of the system clock?
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 ATmega328 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.

   (e) Should an external time base (e.g., crystal oscillator, ceramic resonator) be used when communicating with an external LCD? Explain.

   (f) What is the highest frequency signal that may be generated by the ATmega328? Explain.

   (g) In instrumented balloon flights, the instrumentation package is powered after the balloon is aloft. Develop a program for the ATmega328 that generates a 1-s pulse after 50 min.

   (h) How many simultaneous PWM signals may be generated by the ATmega328?

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.

11. Write functions to draw a circle, diamond, and a sine wave with the X-Y laser control system.

CHAPTER 5

Serial Communication Subsystem

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

- describe the differences between serial and parallel communication;
- provide definitions for key serial communications terminology;
- describe the operation of the Universal Synchronous and Asynchronous Serial Receiver and Transmitter (USART);
- program the USART for basic transmission and reception using C;
- describe the operation of the Serial Peripheral Interface (SPI);
- program the SPI system using C;
- describe the purpose of the Two Wire Interface (TWI); and
- program the TWI system using C.

5.1 OVERVIEW

Serial communication techniques provide a vital link between a microcontroller and certain input devices, output devices, and other microcontrollers. In this chapter, we investigate the serial communication features beginning with a review of serial communication concepts and terminology.[1] We then investigate in turn the following serial communication systems available on the ATmega328 microcontroller: the Universal Synchronous and Asynchronous Serial Receiver and Transmitter (USART), the Serial Peripheral Interface (SPI), and the Two Wire Interface (TWI). We provide guidance on how to program the USART, SPI, and TWI systems using the C programming language.

[1]The sections on serial communication theory were adapted with permission from *Microcontroller Fundamentals for Engineers and Scientists*, S. F. Barrett and D. J. Pack, Morgan & Claypool Publishers, 2006.

5.2 SERIAL COMMUNICATIONS

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. While this is efficient from a time point of view, it requires eight separate lines for the data transfer.

In serial transmission, a byte of data is sent a single bit at a time. Once eight bits have been received at the receiver, the data byte is reconstructed. While this is inefficient from a time point of view, it only requires a line (or two) to transmit the data.

The ATmega328 (UNO R3) is equipped with a host of different serial communication subsystems including the serial USART, the serial peripheral interface or SPI, and the Two-wire Serial Interface (TWI). What all of these systems have in common is the serial transmission of data. Before discussing the different serial communication features aboard these processors, we review serial communication terminology.

5.3 SERIAL COMMUNICATION TERMINOLOGY

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

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 ATmega328, 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 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 edge of the clock. This allows data transfer rates higher than with asynchronous techniques but requires two lines, data and clock, to connect the receiver and transmitter.

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

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.

Non-return to Zero (NRZ) 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 ATmega328 uses a non-return to zero (NRZ) coding standard. In NRZ 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.

The RS–232 Communication Protocol: When serial transmission occurs over a long distance additional techniques may be used to insure 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 while a logic zero is represented by a +12 VDC level. Chips are commonly available (e.g., MAX232) that convert the 5 and 0 V output levels from a transmitter to RS-232 compatible levels and convert back to 5 V and 0 V levels at the receiver. The RS-232 standard also specifies other features for this communication protocol.

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 ATmega328 employs 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 ones 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 insure that parity has not changed, indicating an error, during transmission.

ASCII: The American Standard Code for Information Interchange (ASCII) is a standardized, seven 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 provided in Figure 5.1. For example, the capital letter "G" is encoded in ASCII as 0x47. 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, www.unicode.org, for additional information on this standardized encoding format.

5.4 SERIAL USART

The serial Universal Synchronous and Asynchronous Serial Receiver and Transmitter (USART) provide for full duplex (two way) communication between a receiver and transmitter. This is accomplished by equipping the ATmega328 with independent hardware for the transmitter and receiver. The ATmega328 is equipped with a single USART channel. The USART is typically

Least Significant Digit \ Most Significant Digit	0x0_	0x1_	0x2_	0x3_	0x4_	0x5_	0x6_	0x7_
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 5.1: ASCII Code. The ASCII code is used to encode alphanumeric characters. The “0x” indicates hexadecimal notation in the C programming language.

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. Space does not permit a discussion of these USART enhancements.

The ATmega USART is quite flexible. It has the capability to be set to a variety of data transmission or Baud (bits per second) rates. The USART may also be set for data bit widths of 5–9 bits with one or two stop bits. Furthermore, the ATmega USART 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. We now discuss the operation, programming, and application of the USART. Due to space limitations, we cover only the most basic capability of this flexible and powerful serial communication system.

5.4.1 SYSTEM OVERVIEW

The block diagram for the USART is provided in Figure 5.2. The block diagram may appear a bit overwhelming but realize there are four basic pieces to the diagram: the clock generator,

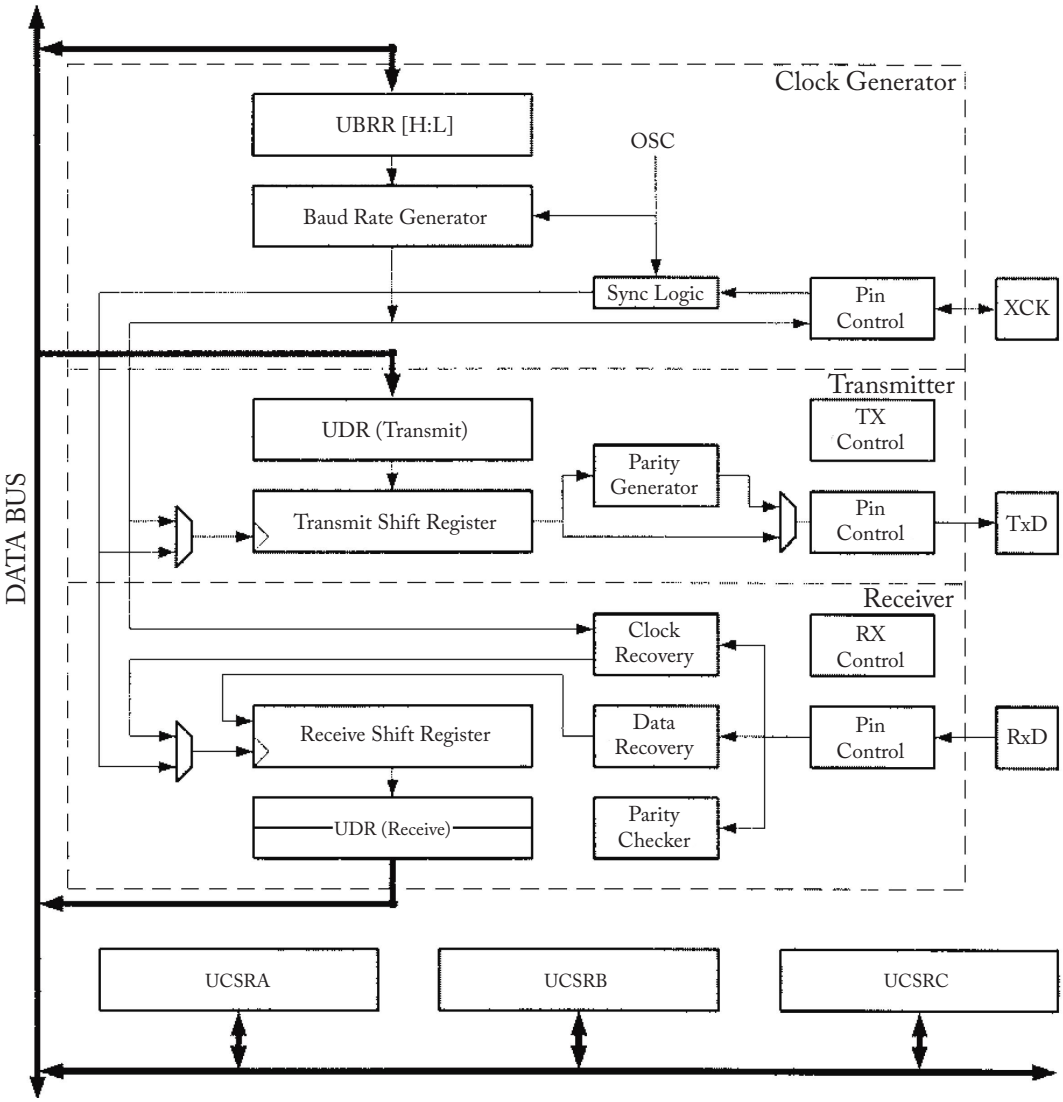


Figure 5.2: Microchip AVR ATmega USART block diagram. (Figure used with permission of Microchip, Inc., [www.microchop.com].)

transmission hardware, receiver hardware, and three control registers (UCSRA, UCSBR, and UCSRC). We discuss each in turn.

5.4.1.1 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 UBRR[H:L] and several additional dividers to set the Baud rate. For the asynchronous normal mode (U2X bit = 0), the Baud Rate is determined using the following expression:

$$\text{Baud rate} = (\text{system clock frequency})/(16(\text{UBRR} + 1)),$$

where UBRR is the contents of the UBRRH and UBRRL registers (0 to 4095). Solving for UBRR yields:

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

5.4.1.2 USART Transmitter

The USART transmitter consists of a Transmit Shift Register. The data to be transmitted is loaded into the Transmit Shift Register via the USART I/O Data Register (UDR). 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 is 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 UDRE and the TXC. The USART Data Register Empty (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 (UCSRA). The UDRE bit is cleared by writing to the USART I/O Data Register (UDR). The Transmit Complete (TXC) Flag bit is set to logic one 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 one to it.

5.4.1.3 USART Receiver

The USART Receiver is virtually identical to the USART Transmitter except for the direction of the data flow is reversed. Data is 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 one when unread data exists in the receive buffer.

5.4.1.4 USART Registers

In this section, we discuss the register settings for controlling the USART system. We have already discussed the function of the USART I/O Data Register (UDR) and the USART Baud Rate Registers (UBRRH and UBRRL). **Note:** The USART Control and Status Register C (UCSRC) and the USART Baud Rate Register High (UBRRH) are assigned to the same I/O location in the memory map. The URSEL bit (bit 7 of both registers) determine which register is being accessed. The URSEL bit must be one when writing to the UCSRC register and zero when writing to the UBRRH register.

Note: As previously mentioned, the ATmega328 is equipped with a single USART. The registers to configure the ATmega328 is provided in Figure 5.3.

USART Control and Status Register A (UCSRA) The UCSRA register contains the RXC, TXC, and the UDRE bits. The function of these bits have already been discussed.

USART Control and Status Register B (UCSRB) The UCSRB register contains the Receiver Enable (RXEN) bit and the Transmitter Enable (TXEN) bit. These bits are the "on/off" switch for the receiver and transmitter, respectively. The UCSRB register also contains the UCSZ2 bit. The UCSZ2 bit in the UCSRB register and the UCSZ[1:0] bits contained in the UCSRC register together set the data character size.

USART Control and Status Register C (UCSRC) The UCSRC register 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 UCSRC contains the following bits:

- USART Mode Select (UMSEL) – 0: asynchronous operation, 1: synchronous operation;
- USART Parity Mode (UPM[1:0]) – 00: no parity, 10: even parity, 11: odd parity;
- USART Stop Bit Select (USBS) – 0: 1 stop bit, 1: 2 stop bits; and
- USART Character Size (data width). (UCSZ[2:0]) – 000: 5-bit, 001: 6-bit, 010: 7-bit, 011: 8-bit, 111: 9-bit.

5.5 SYSTEM OPERATION AND PROGRAMMING IN C

The basic activities of the USART system consist of initialization, transmission, and reception. These activities are summarized in Figure 5.4. 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

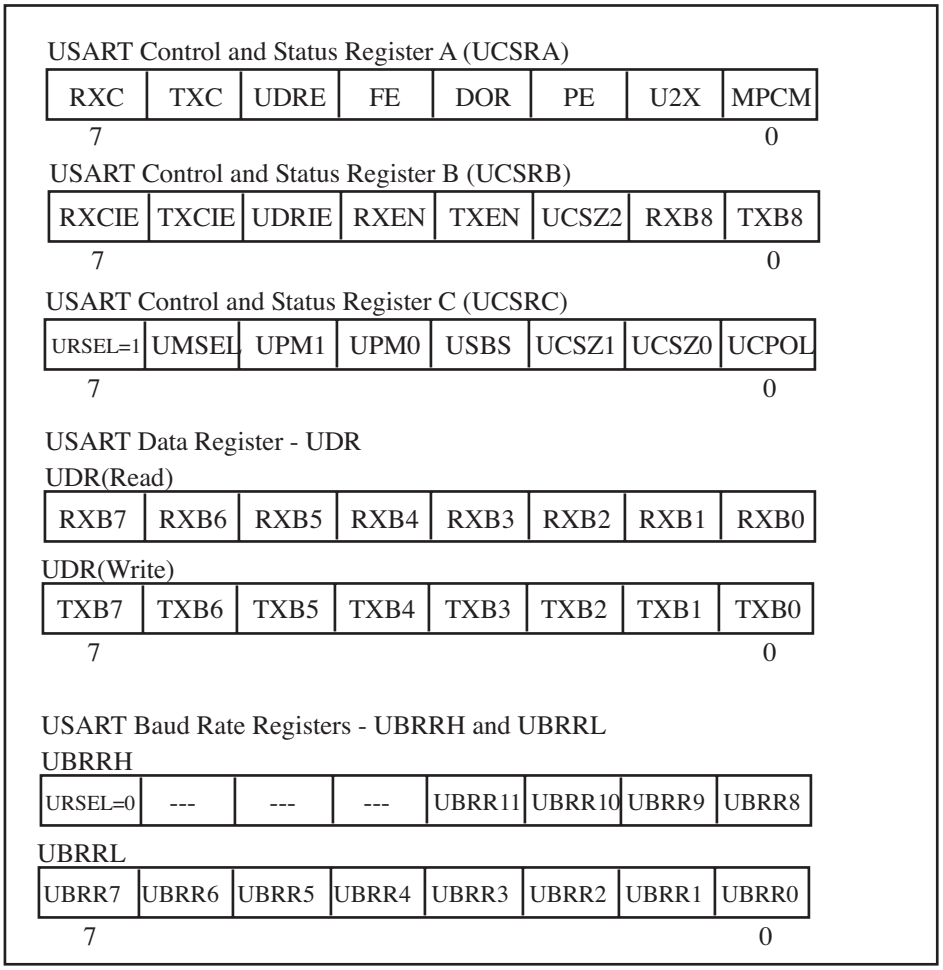


Figure 5.3: ATmega328 USART Registers.

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

Note: As previously mentioned, the ATmega328 is equipped with a single USART channel. The registers to configure the ATmega328 is provided in Figure 5.3.

To program the USART, we implement the flow diagrams provided in Figure 5.4. In the sample code provided, we assume the ATmega328 is operating at 10 MHz, and we desire a Baud Rate of 9600, asynchronous operation, no parity, one stop bit, and eight data bits.

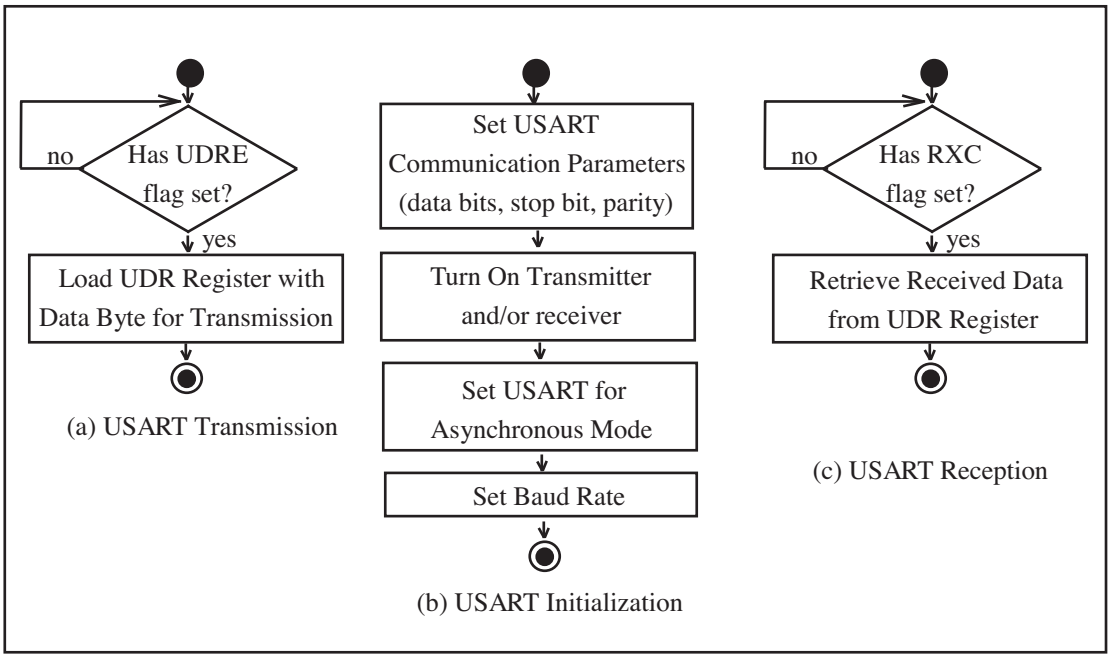


Figure 5.4: USART Activities.

To achieve 9600 Baud with an operating frequency of 10 MHz requires that we set the UBRR registers to 64 which is 0x40.[2]

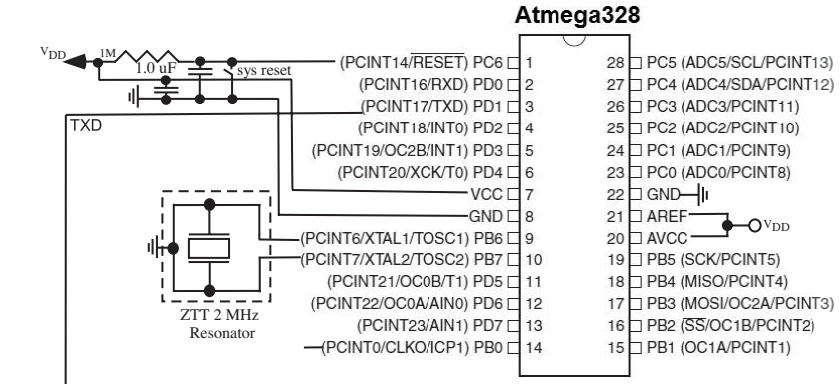
```
//************************************************************
//USART_init: initializes the USART system
//************************************************************

void USART_init(void)
{
UCSRA = 0x00;                 //control register initialization
UCSRB = 0x08;                 //enable transmitter
UCSRC = 0x86;                 //async, no parity,
                              //1 stop bit, 8 data bits
                              //Baud Rate initialization
UBRRH = 0x00;
UBRRL = 0x40;
```

[2]Chapter examples were originally developed for the ATmega164 and provided in *Microchip AVR Microcontroller Primer: Programming and Interfacing*, 3rd ed., S. F. Barrett and D. J. Pack, 2019. The examples were adapted with permission for the ATmega328.

```
}

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

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

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

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

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

5.5.1 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 ATmega328. The interface is shown in Figure 5.5. The ATmega328 USART channel 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.



Prefix	Command	Description
--	--	display character (0x00-0xff)
0xFE	0x41	display on
0xFE	0x45	position - line 1: 0x00 to 0x0F - line 2: 0x40 to 0x4F
0xFE	0x46	cursor home
0xFE	0x51	clear screen
0xFE	0x52	set contrast (1 to 50)
0xFE	0x53	set background brightness (1 to 8)

Figure 5.5: Serial LCD connections.

In this specific example a 9600 BAUD rate is required by the LCD. The ATmega328 is clocked by a 2-MHz ceramic resonator. The UBRR registers (UBRR0H, UBRR0L) must be set to 12 (0x0c) to achieve the desired 9600 BAUD (bits per second rate). The ASCII representation of "G" is shown in Figure 5.6.

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

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

start bit	1	1	1	0	0	0	1	parity bit	stop bit

ASCII representation of "G" (0x47, P100_0111) at 9600 BAUD.
The oscilloscope settings are 1 V/div horizontal, 100 us/div vertical.

Figure 5.6: ASCII representation of "G". Signal captured with a Siglent SDS 1104 X-E digital storage oscilloscope, 4-channel, 100 MHz.

```
//include file(s) for JumpStart C for AVR Compiler****************
#include<iom328pv.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);
void lcd_print_string(char str[]);
void move_LCD_cursor(unsigned char position);
```

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

USART_init();
LCD_init();

while(1)
  {
  USART_transmit(0xFE);
  USART_transmit(0x46);                //cursor to home

  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)
{
UCSR0A = 0x00;                       //control register init
UCSR0B = 0x08;                       //enable transmitter
UCSR0C = 0x06;                       //async, no parity,
                                     //1 stop bit, 8 data bits
                                     //Baud Rate initialization
UBRR0H = 0x00; UBRR0L = 0x0c;        //9600 BAUD, 2 MHz clock
                                     //divider set to 12
}

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

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

void USART_transmit(unsigned char data)
{
while((UCSR0A & 0x20)==0x00) //wait for UDRE flag
  {
  ;
  }
UDR0 = data;                          //load data to UDR for tx
}

//*****************************************************************
//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);
}

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

5.5.2 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 ATmega328 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 5.7. Driver installation instructions for the cable is provided at the Sparkfun website.

Messages sent from the ATmega328 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 ATmega328 to the support computer.

```
//*************************************************************
//usart_to_pc: provides one way communication from ATmega328
//USART0 back to a host (PC or laptop).
//In this example the ATmega328 was clocked by a 2.0 MHz
//external ceramic resonator.
//*************************************************************

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

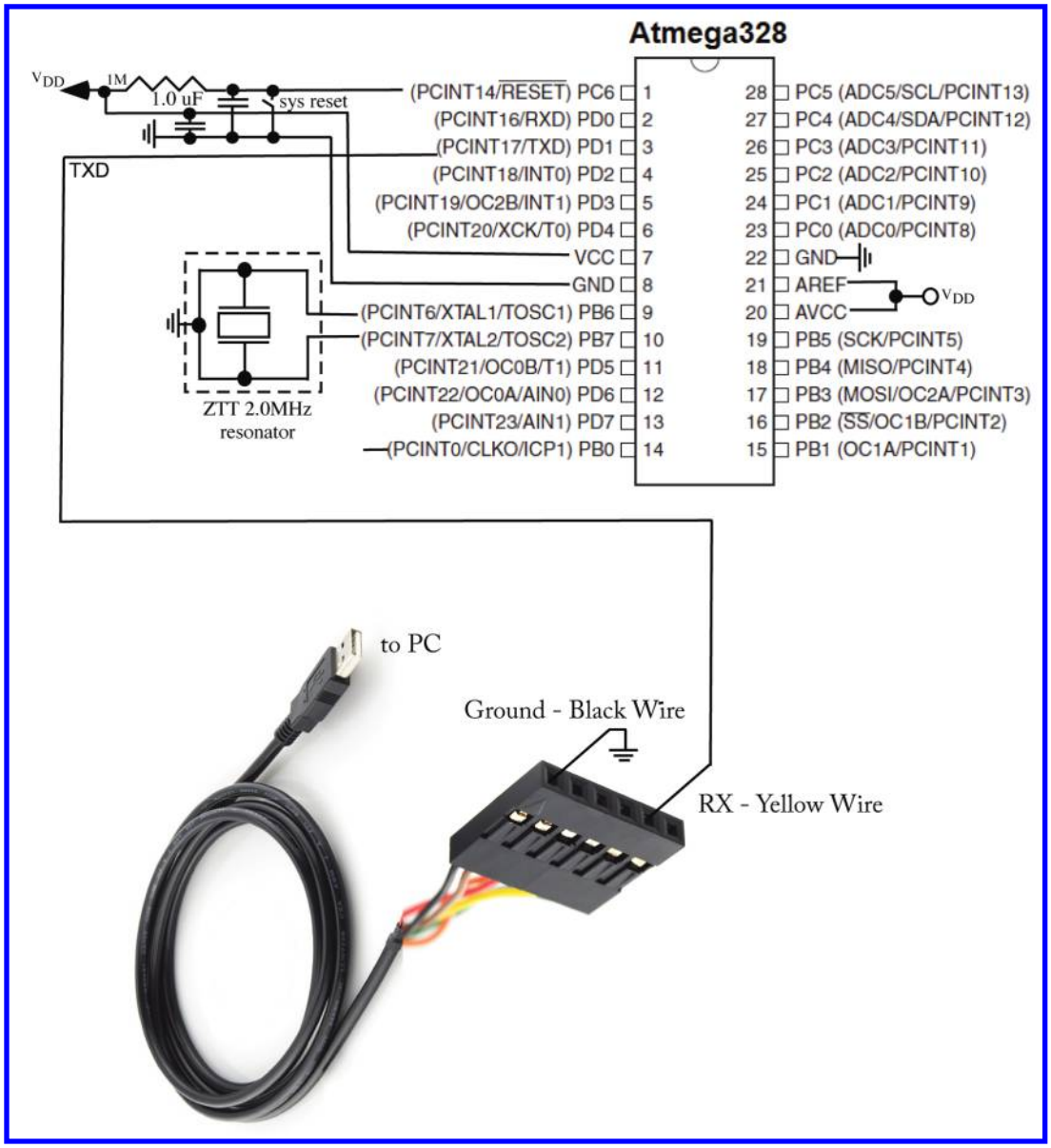


Figure 5.7: USART to computer communication link [www.sparkfun.com].

```
//include file(s) for JumpStart C for AVR Compiler****************
#include<iom328pv.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_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
}

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

```
//delay_100ms: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 2.0 MHz
//  - 100 ms delay requires 200,000 clock cycles
//  - nop requires 1 clock cycle to execute
//*****************************************************************

void delay_100ms(void)
{
unsigned int i,j;

for(i=0; i < 200; i++)
  {
  for(j=0; j < 1000; j++)
    {
    asm("nop");
    }
  }
}

//*****************************************************************
//delay_1s: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 2.0 MHz
//  - 100 ms delay requires 200,000 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();
  }
}

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

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

void USART_init(void)
{
UCSR0A = 0x00;                          //control register init
UCSR0B = 0x08;                          //enable transmitter
UCSR0C = 0x06;                          //async, no parity,
                                        //1 stop bit, 8 data bits
                                        //Baud Rate initialization
UBRR0H = 0x00; UBRR0L = 0x0c;           //9600 BAUD, 2 MHz clock
                                        //divider set to 12 (0x0c)
}

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

void USART_transmit(unsigned char data)
{
while((UCSR0A & 0x20)==0x00) //wait for UDRE flag
  {
  ;
  }
UDR0 = data;                             //load data to UDR for tx
}

//*****************************************************************
//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);
}

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

5.5.3 SERIAL PERIPHERAL INTERFACE (SPI)

The ATmega Serial Peripheral Interface or SPI 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 to 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 or a DAC could be added to the microcontroller using the SPI system.

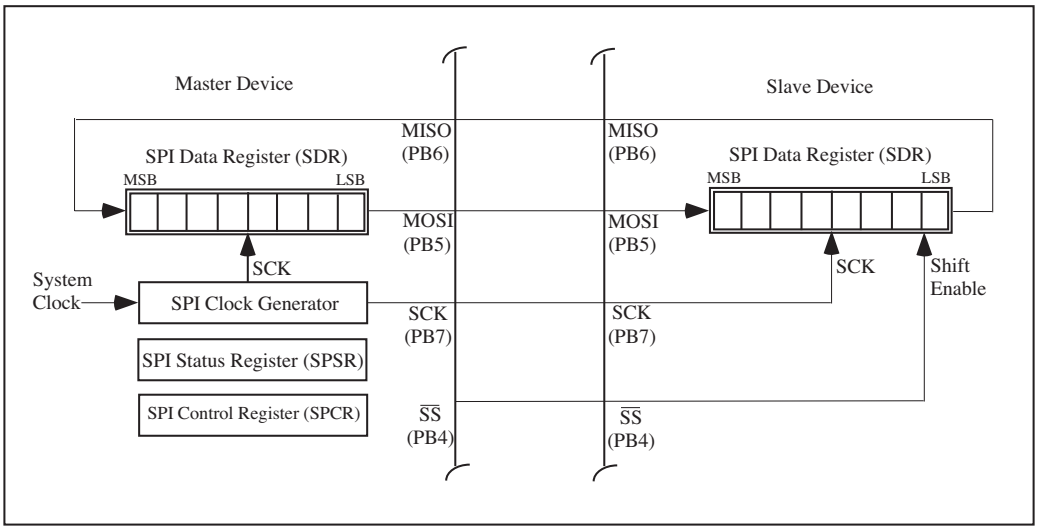


Figure 5.8: SPI Overview.

5.5.3.1 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 5.8. The transmitter is designated the master since it is providing 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 ($\overline{SS}$) line low. When the $\overline{SS}$ 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 is 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 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.

5.5.3.2 Registers

The registers for the SPI system are provided in Figure 5.9. We will discuss each one in turn.

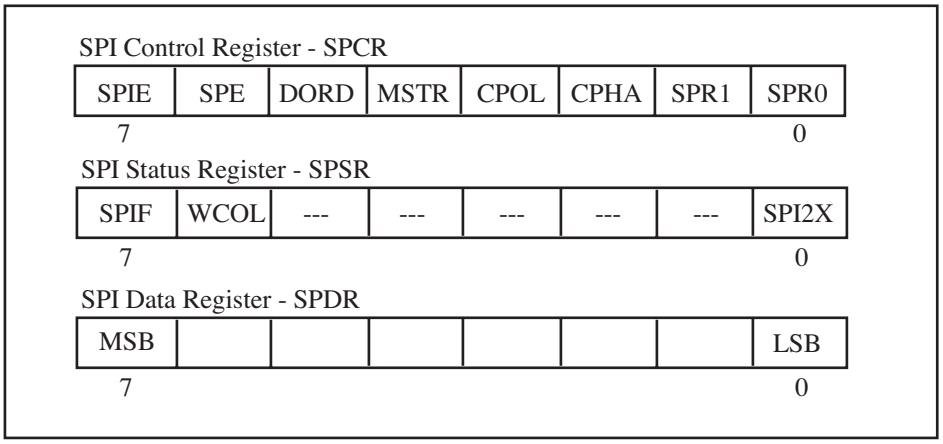


Figure 5.9: SPI Registers [www.microchip.com].

SPI Control Register (SPCR) 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 one turns the system on and logic zero 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 one, the least significant bit (LSB) of the SPI Data Register (SPDR) is transmitted first. When the DORD bit is set to zero 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 one) or slave (logic zero).
- The Clock Polarity (CPOL) bit allows determines the idle condition of the SCK pin. When CPOL is one, SCK will idle logic high; whereas, when CPOL is zero, SCK will idle logic zero.
- 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 bits SPR[1:0]

and the Double SPI Speed Bit (SPI2X) are used to set the division factor. The following divisions may be selected using SPI2X, SPR1, SPR0:

- 000: SCK = system clock/4
- 001: SCK = system clock/16
- 010: SCK = system clock/64
- 011: SCK = system clock/1284
- 100: SCK = system clock/2
- 101: SCK = system clock/8
- 110: SCK = system clock/32
- 111: SCK = system clock/64

SPI Status Register (SPSR) The SPSR contains the SPI Interrupt Flag (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 SPI Data Register (SPDR). The SPSR also contains the SPI2X bit used to set the SCK frequency.

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

5.6 SPI PROGRAMMING IN THE ARDUINO DEVELOPMENT ENVIRONMENT

The ADE provides the "shiftOut" command to provide ISP style serial communications [www.Arduino.cc]. The shiftOut command requires four parameters.

- **dataPin:** the Arduino UNO R3 DIGITAL pin to be used for serial output.
- **clockPin:** the Arduino UNO R3 DIGITAL pin to be used for the clock.
- **bitOrder:** indicates whether the data byte will be sent most significant bit first (MSBFIRST) or least significant bit first (LSBFIRST).
- **value:** the data byte that will be shifted out.

To use the shiftOut command, the appropriate pins are declared as output using the pinMode command in the setup() function. The shiftOut command is then called at the appropriate place within the loop() function using the following syntax:

```
shiftOut(dataPin, clockPin, LSBFIRST, value);
```

As a result of the this command, the value specified will be serially shifted out of the data pin specified, least significant bit first, at the clock rate provided at the clock pin.

5.7 SPI PROGRAMMING IN C

To program the SPI system in C, 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. **Note:** For proper SPI operation the slave select pin (PB2) must be set to output even if not used [www.avrfreaks.com].

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

void spi_init(unsigned char control)
{
DDRB = 0x2c;            //Set SCK (PB5), MOSI (PB3), /SS (PB2)
                        //for output, others to input
SPCR = 0x53;            //Configure SPI Control Register (SPCR)
                        //SPIE:0,SPE:1,DORD:0,MSTR:1
                        //CPOL:0,CPHA:0,SPR:1,SPR0:1
                        //Divide clock by 128
}

//*****************************************************************
//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;
}

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

5.7.1 EXAMPLE: LED STRIP

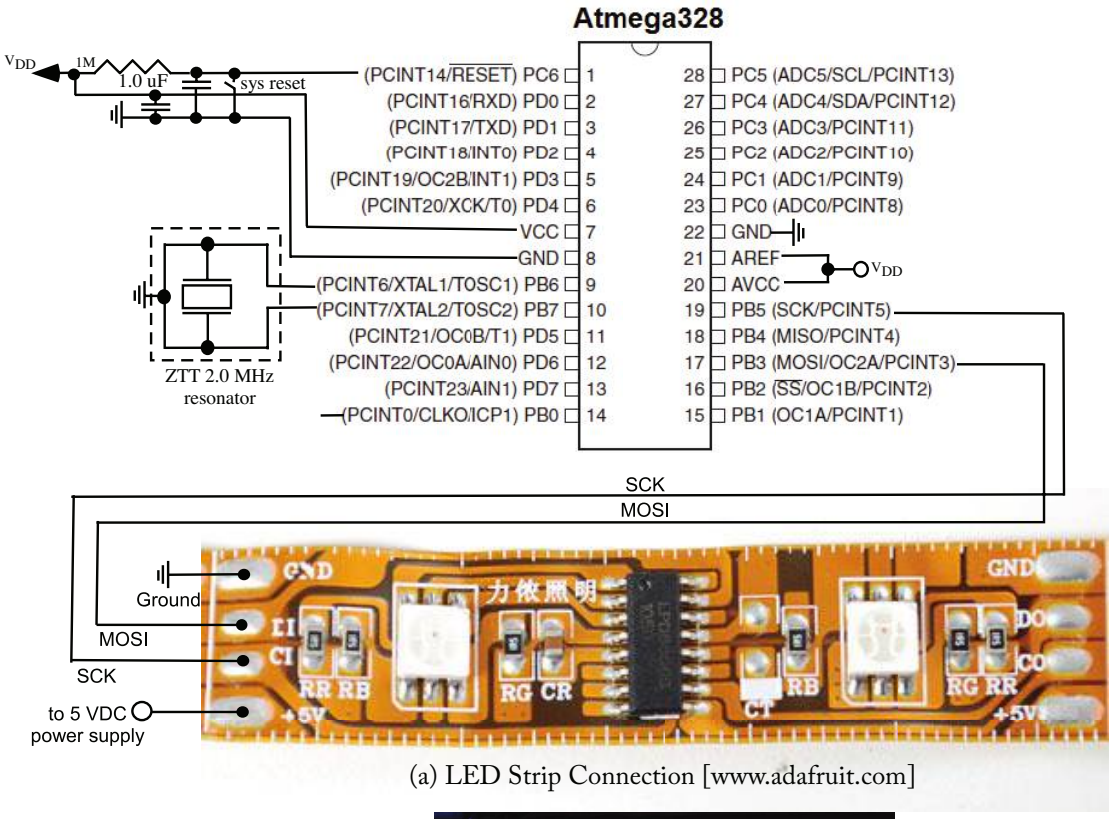
LED strips may be used for motivational (fun) optical displays, games, or for instrumentation-based applications. In this example we control an LPD8806-based LED strip. We use a 1-m, 32-RGB LED strip available from Adafruit (#306) for approximately $30 USD [www.adafruit.com].

The red, blue, and green component of each RGB LED is independently set using an eight-bit code. The most significant bit (MSB) is logic one followed by seven bits to set the LED intensity (0–127). The component values are sequentially shifted out of the ATmega328 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].

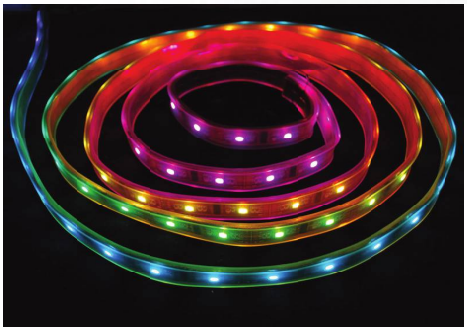
Only four connections are required between the ATmega328 and the LED strip as shown in Figure 5.10. The connections are color coded: red-power, black-ground, yellow-data, and green-clock. The ATmega328 is equipped with a single SPI channel. This channel is used to program the ATmega328 using ISP techniques. 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 5 V 2 A (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.
- 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.



(a) LED Strip Connection [www.adafruit.com]



(b) LED Strip by the Meter [www.adafruit.com]

Figure 5.10: ATmega328 controlling LED strip [www.adafruit.com].

```
//*****************************************************************
//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:  ATmega328: MOSI PORTB[3], pin 17
// -            CLK:  ATmega328: SCK  PORTB[5], pin 19
//
//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<iom328pv.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 = 0x2c;           //Set SCK (PB5), MOSI (PB3), /SS (PB2)
                       //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 2.0 MHz
//  - 100 ms delay requires 200,000 clock cycles
//  - nop requires 1 clock cycle to execute
//*****************************************************************

void delay_100ms(void)
{
unsigned int i,j;

for(i=0; i < 200; i++)
  {
  for(j=0; j < 1000; j++)
    {
    asm("nop");
    }
  }
}

//*****************************************************************
//delay_1s: inaccurate, yet simple method of creating delay
//  - processor clock: ceramic resonator at 2.0 MHz
//  - 100 ms delay requires 200,000 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();
  }
}

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

5.8 TWO-WIRE SERIAL INTERFACE

The TWI subsystem allows the system designer to connect a number of TWI configured 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 [4]].

An overview of the TWI system is shown in Figure 5.11. 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.

The TWI system is configured and controlled using a series of registers shown in Figure 5.11. Details on specific bit settings are provided in the Microchip ATmega328 datasheet and will not be duplicated here [www.microchip.com]. These include:

- TWBR: TWI Bit Rate Register

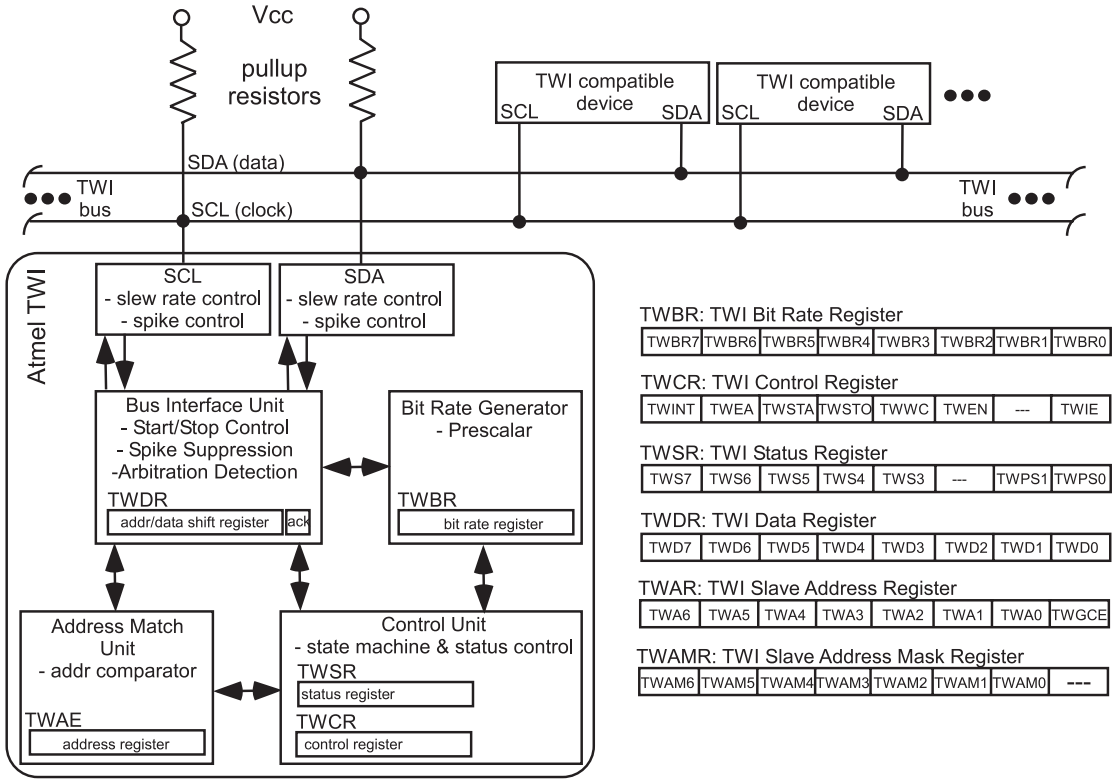


Figure 5.11: TWI system overview [Microchip [5]].

- 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 5.12. On the left-hand side of the figure are the actions required 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.

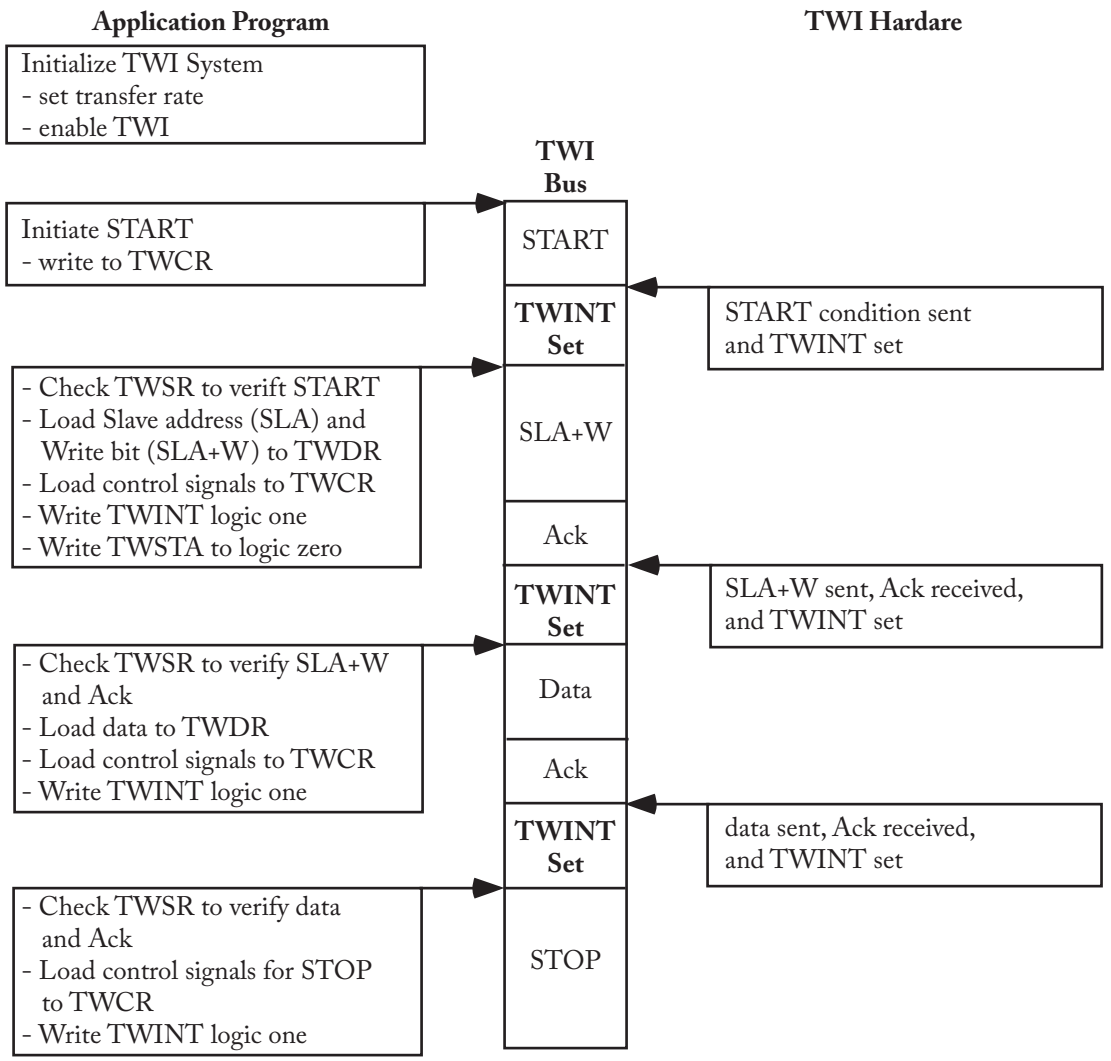
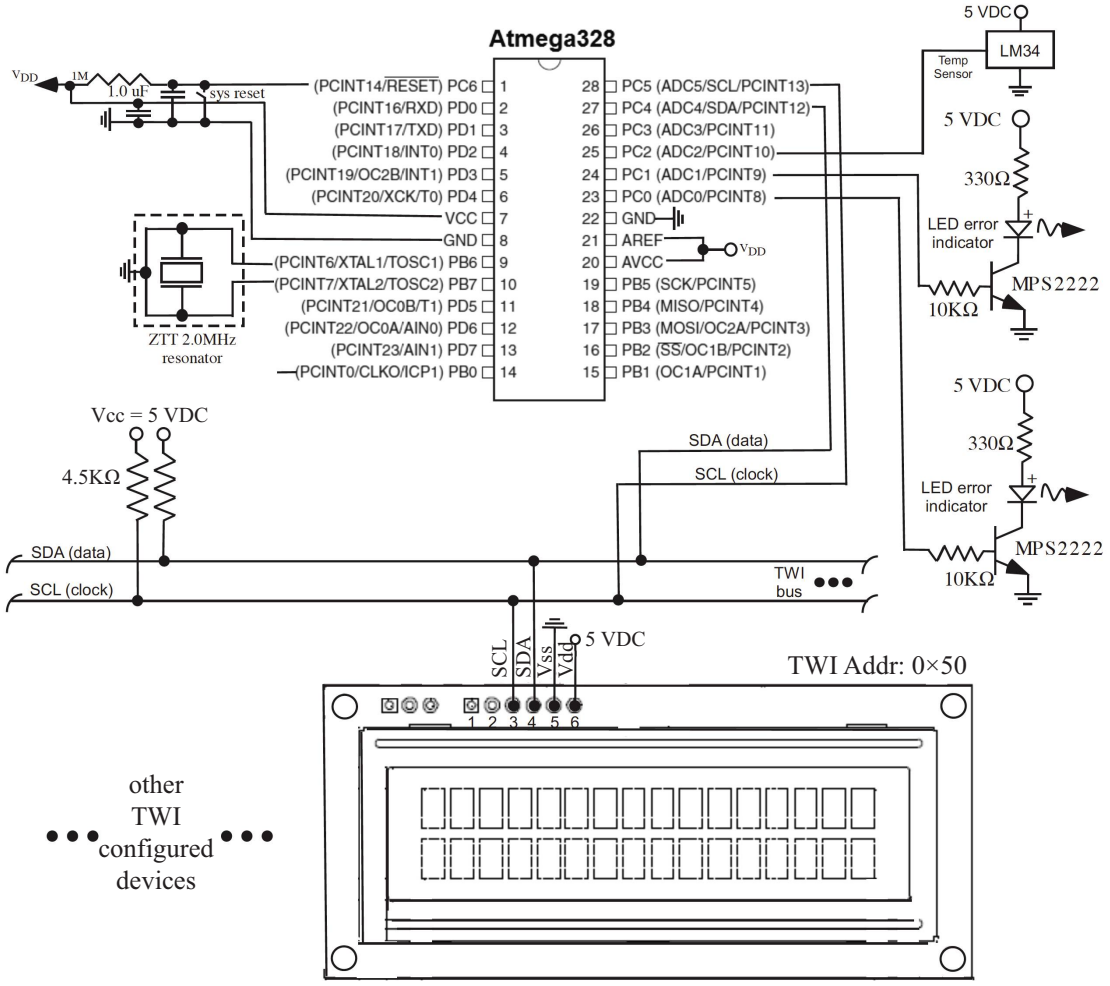


Figure 5.12: TWI operations [www.microchip.com].

5.8.1 EXAMPLE: TWI-COMPATIBLE LCD

In this example a TWI compatible LCD is used to display temperature data from an LM34 temperature sensor, as shown in Figure 5.13. Note how the LCD is connected to the TWI bus via the SDA and SCL pins.

```
//*************************************************************
//twi2.c
//
//ATmega328 is clocked by a 2.0 MHz ceramic resonator
//
//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<iom328pv.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
```



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

Figure 5.13: TWI connecting sensor to LCD [www.sparkfun.com].

```
//clock specifications
#define ceramic_res_freq  2000000UL   //ATmega328 operating freq
#define scl_clock         100000L     //desired TWI bus freq

//peripheral device addresses
#define LCD_twi_addr      0x50        //addr - LSB 0 for write

//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);
void InitADC( void);
unsigned int ReadADC(unsigned char channel);
void temperatureToLCD(unsigned int);
void delay_10ms(void);
void delay_100ms(void);

void main(void)
{
unsigned int temp_int;

initialize_ports();
twi_initialize();
InitADC();
LCD_init();
delay_100ms();

while(1)
  {
  twi_send_byte(LCD_twi_addr, 0xFE);
  twi_send_byte(LCD_twi_addr, 0x51); //clear LCD
  delay_10ms();

  twi_send_byte(LCD_twi_addr, 0xFE);
```

```
  twi_send_byte(LCD_twi_addr, 0x46); //cursor to home
  lcd_print_string("Temp:");
  delay_10ms();

  temp_int = ReadADC(0x02);          //read temp from LM34

                                     //move cursor line 2, pos 0
  move_LCD_cursor(0x40);
  temperatureToLCD(temp_int);
  delay_100ms();
  delay_100ms();
  }
}

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

void initialize_ports(void)
{
DDRB=0xff;                          //PORTB[7:0] as output
PORTB=0x00;                         //initialize low

DDRC=0xfb;                          //set PORTC as output, C[2] input
PORTC=0x00;                         //initialize low

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

}

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

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

```
if(error_number == 1)             //Error 01
  PORTC = 0x01;
else if(error_number == 2)        //Error 10
  PORTC = 0x02;
else if(error_number == 3)        //Error 11
  PORTC = 0x03;
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;
  delay_10ms();
  }
}

//*****************************************************************
//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);
delay_10ms();
}

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

void twi_initialize(void)
{
//set twi frequency to 100 kHz with ceramic
//resonator frequency at 2.0 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);
}

//*****************************************************************
//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
}

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

void temperatureToLCD(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);

twi_send_byte(LCD_twi_addr, ((unsigned char)(tens)+48) );
twi_send_byte(LCD_twi_addr, ((unsigned char)(ones)+48));
twi_send_byte(LCD_twi_addr, '.');
```

```
twi_send_byte(LCD_twi_addr, ((unsigned char)(tenths)+48));
twi_send_byte(LCD_twi_addr, 'F');

}

//*************************************************************
//delay_10ms: inaccurate, yet simple method of creating delay
// - processor clock: ceramic resonator at 2 MHz
// - nop requires 1 clock cycle to execute
// - 10 ms delay requires 20,000 clock cycles
//*************************************************************

void delay_10ms(void)
{
unsigned int i,j;

for(i=0; i < 20; i++)
  {
  for(j=0; j < 1000; j++)
    {
    asm("nop"); //inline assembly
    } //nop: no operation
  } //requires 1 clock cycle
}

//*************************************************************
//delay_10ms: inaccurate, yet simple method of creating delay
// - processor clock: ceramic resonator at 2 MHz
// - nop requires 1 clock cycle to execute
// - 100 ms delay requires 200,000 clock cycles
//*************************************************************

void delay_100ms(void)
{
unsigned int i,j;

for(i=0; i < 200; i++)
  {
  for(j=0; j < 1000; j++)
```

```
    {
    asm("nop"); //inline assembly
    } //nop: no operation
  } //requires 1 clock cycle
}

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

This example demonstrated how to write to a TWI configured device. The TWI system may also be used to read from TWI configured peripherals. Microchip provides a library of TWI functions [www.microchip.com].

5.9 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 USART, SPI, and TWI serial communication systems. We also provided basic code examples to communicate with the USART, SPI, and TWI systems.

5.10 REFERENCES

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

[2] S. F. Barrett and D. J. Pack. *Atmel® AVR Microcontroller Primer Programming and Interfacing*, Morgan & Claypool Publishers, 2008. DOI: 10.2200/s00100ed1v01y200712dcs015.

[3] S. F. Barrett. *Embedded Systems Design with the Atmel® AVR Microcontroller*, Morgan & Claypool Publishers, 2010. DOI: 10.2200/s00138ed1v01y200910dcs024.

[4] *The I2C—Bus Specification*. Version 2.1, Philips Semiconductor, January 2000. 172

[5] *Microchip ATmega328 PB AVR Microcontroller with Core Independent Peripherals and Pico Power Technology DS40001906C*, Microchip Technology Incorporation, 2018. www.microchip.com 173

[6] S. F. Barrett and D. J. Pack. *Microchip AVR Microcontroller Primer: Programming and Interfacing*, 3rd ed., Morgan & Claypool Publishers, 2019.

5.11 CHAPTER PROBLEMS

1. Summarize the differences between parallel and serial bit stream conversion techniques.
2. Summarize the differences between 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 SPI flags.
5. Draw a block diagram of the TWI system, label all key registers, and all keys TWI flags.
6. If an ATmega328 microcontroller is operating at 12-MHz, what is the maximum transmission rate for the USART and the SPI?
7. What is the ASCII encoded value for "ATmega328"?
8. Draw the schematic of a system consisting of two ATmega328s that will exchange data via the SPI system.
9. Write the code to implement the system described in the question above.
10. Modify the TWI example provided in the chapter to read the temperature from a TMP 102 digital temperature sensor and display the result on an LCD. Provide a schematic and program.
11. Are there any limitations to connecting multiple devices to a TWI configured network? Explain.
12. It is desired to equip the ATmega328 with eight channels of digital-to-analog conversion. What serial communication system should be employed. Provide a detailed design.
13. Research the BlinkM Smart LED. Provide a schematic to connect the LED to the ATmega328.
14. Write a program to implement the interface between the ATmega328 and the BlinkM Smart LED.
15. Research Sparkfun's 6.5-inch 7-segment displays. Provide a schematic to connect the displays to the ATmega328. Write a program to implement the interface between the ATmega328 and the displays.

CHAPTER 6

Interrupt Subsystem

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

- understand the need of a microcontroller for interrupt capability;
- describe the general microcontroller interrupt response procedure;
- describe the ATmega328 interrupt features;
- properly configure and program an interrupt event for the ATmega328 in C;
- properly configure and program an interrupt event for the Arduino UNO R3 using built-in features of the ADE;
- use the interrupt system to implement a real time clock; and
- employ the interrupt system as a component in an embedded system.

6.1 OVERVIEW

A microcontroller normally executes instructions in an orderly fetch-decode-execute sequence as dictated by a user-written program as shown in Figure 6.1. However, the microcontroller must be equipped to handle unscheduled (although planned), higher priority events that might occur inside or outside the microcontroller. To process such events, a microcontroller requires an interrupt system.[1]

The interrupt system onboard a microcontroller allows it to respond to higher priority events. Appropriate responses to these events may be planned, but we do not know when these events 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 complete, the microcontroller will resume processing where it left off before the interrupt event occurred.

In this chapter, we discuss the ATmega328 interrupt system. We provide several examples on how to program an interrupt in C and also using the built-in features of the ADE.

[1]The sections on interrupt theory were adapted with permission from *Microcontroller Fundamentals for Engineers and Scientists*, S. F. Barrett and D. J. Pack, Morgan & Claypool Publishers, 2006.

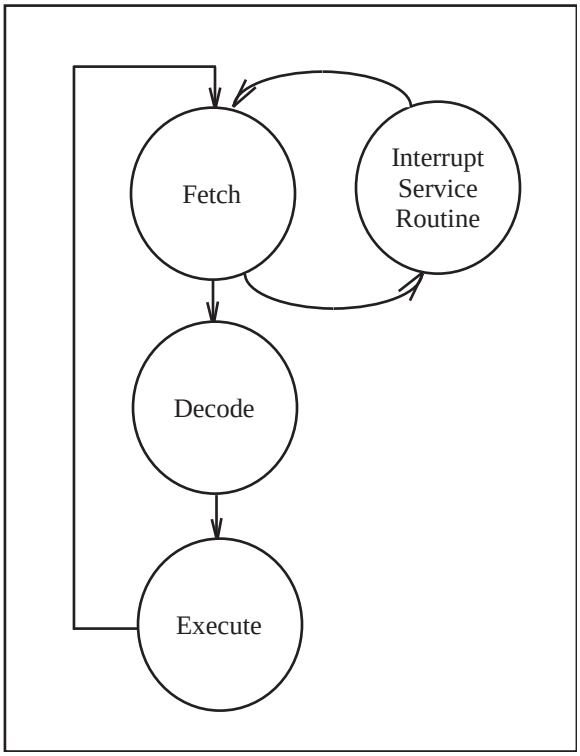


Figure 6.1: Microcontroller Interrupt Response.

6.1.1 ATMEGA328 INTERRUPT SYSTEM

The ATmega328 is equipped with a powerful and flexible complement of 26 interrupt sources. Two of the interrupts originate from external interrupt sources while the remaining 24 interrupts support the efficient operation of peripheral subsystems aboard the microcontroller. The ATmega328 interrupt sources are shown in Figure 6.2. 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 4) and INT1 (pin 5). The remaining interrupt sources are internal to the ATmega328.

6.1.2 GENERAL INTERRUPT RESPONSE

When an interrupt occurs, the microcontroller completes the current instruction, stores the address of the next instruction on the stack, and starts executing instructions in the designated interrupt service routine (ISR) corresponding to the particular interrupt source. It also turns off the interrupt system to prevent further interrupts while one is in progress. The execution of the ISR is performed by loading the beginning address of the interrupt service routine specific

Vector	Address	Source	Definition	AVR-GCC ISR name
1	0x0000	RESET	External pin, power-on reset, brown-out reset, watchdog system reset	
2	0x0002	INT0	External interrupt request 0	INT0_vect
3	0x0004	INT1	External interrupt request 1	INT1_vect
4	0x0006	PCINT0	Pin change interrupt request 0	PCINT0_vect
5	0x0008	PCINT1	Pin change interrupt request 1	PCINT1_vect
6	0x000A	PCINT2	Pin change interrupt request 2	PCINT2_vect
7	0x000C	WDT	Watchdog time-out interrupt	WDT_vect
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A	TIMER2_COMPA_vect
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B	TIMER2_COMPB_vect
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow	TIMER2_OVF_vect
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event	TIMER1_CAPT_vect
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A	TIMER1_COMPA_vect
13	0x0018	TIMER1 COMPB	Timer/Counter1 Compare Match B	TIMER1_COMPB_vect
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow	TIMER1_OVF_vect
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A	TIMER0_COMPA_vect
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B	TIMER0_COMPB_vect
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow	TIMER0_OVF_vect
18	0x0022	SPI, STC	SPI Serial Transfer Complete	SPI_STC_vect
19	0x0024	USART, RX	USART Rx Complete	USART_RX_vect
20	0x0026	USART, UDRE	USART, Data Register Empty	USART_UDRE_vect
21	0x0028	USART, TX	USART, Tx Complete	USART_TX_vect
22	0x002A	ADC	ADC Conversion Complete	ADC_vect
23	0x002C	EE READY	EEPROM Ready	EE_READY_vect
24	0x002E	ANALOG COMP	Analog Comparator	ANALOG_COMP_vect
25	0x0030	TWI	2-wire Serial Interface	TWI_vect
26	0x0032	SPM READY	Store Program Memory Ready	SPM_ready_vect

Figure 6.2: Microchip AVR ATmega328 Interrupts. (Adapted from figure used with permission of Microchip, Inc.)

for that interrupt into the program counter. The interrupt service routine will then commence. Execution of the ISR continues until the return from interrupt instruction (reti) is encountered. Program control then reverts back to the main program.

6.2 INTERRUPT PROGRAMMING OVERVIEW

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

- Ensure the interrupt service routine for a specific interrupt is tied to the correct interrupt vector address, which points to the starting address of the interrupt service routine.
- Ensure the interrupt system has been globally enabled. This is accomplished with the assembly language instruction SEI.
- Ensure the specific interrupt subsystem has been locally enabled.
- Ensure the registers associated with the specific interrupt have been configured correctly.

In the examples that follow, we illustrate how to accomplish these steps. With several different compilers.

6.3 PROGRAMMING ATMEGA328 INTERRUPTS IN C AND THE ARDUINO DEVELOPMENT ENVIRONMENT

In this section, we provide representative examples of writing interrupts. We provide both an externally generated interrupt event and also one generated from within the microcontroller. For each type of interrupt, we illustrate how to program it in C using both the Microchip AVR Visual Studio gcc compiler and the ImageCraft JumpStart C for AVR compiler and also with the ADE built-in features.

6.3.1 MICROCHIP AVR VISUAL STUDIO GCC COMPILER INTERRUPT TEMPLATE

The Microchip 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 fifth column of Figure 6.2. 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 ATmega328
//*****************************************************************

#include <avr/io.h>
#include <avr/interrupt.h>

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

ISR(vector_identifier)
{

:
//programmer written interrupt specific actions
:

}

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

6.3.2 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 6.2. 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<iom328pv.h>                    //contains reg definitions
```

```
#pragma interrupt_handler timer_handler:17

void timer_handler(void)
{

:
//programmer written interrupt specific actions
:


}
//*****************************************************************
```

6.3.3 EXTERNAL INTERRUPT PROGRAMMING-ATMEGA328

The external interrupts INT0 (pin 4) and INT1 (pin 5) trigger an interrupt within the ATmega328 when an user-specified external event occurs at the pin associated with the specific interrupt. Interrupts INT0 and INT1 may be triggered with a falling or rising edge or a low-level signal. The specific settings for each interrupt is provided in Figure 6.3.

6.3.3.1 Programming External Interrupts in C-ImageCraft

Provided below is the code snapshot to configure an interrupt for INT0. In this specific example, an interrupt will occur when a positive edge transition occurs on the ATmega328 INT0 external interrupt pin.

```
//*****************************************************************
//interrupt handler definition
#pragma interrupt_handler int0_ISR:2

//function prototypes
void int0_ISR(void);
void initialize_interrupt0(void);
//*****************************************************************

//The following function call should be inserted in the main
//program to initialize the INT0 interrupt to respond to a positive
//edge trigger.  This function should only be called once.

:
initialize_interrupt_int0();
```

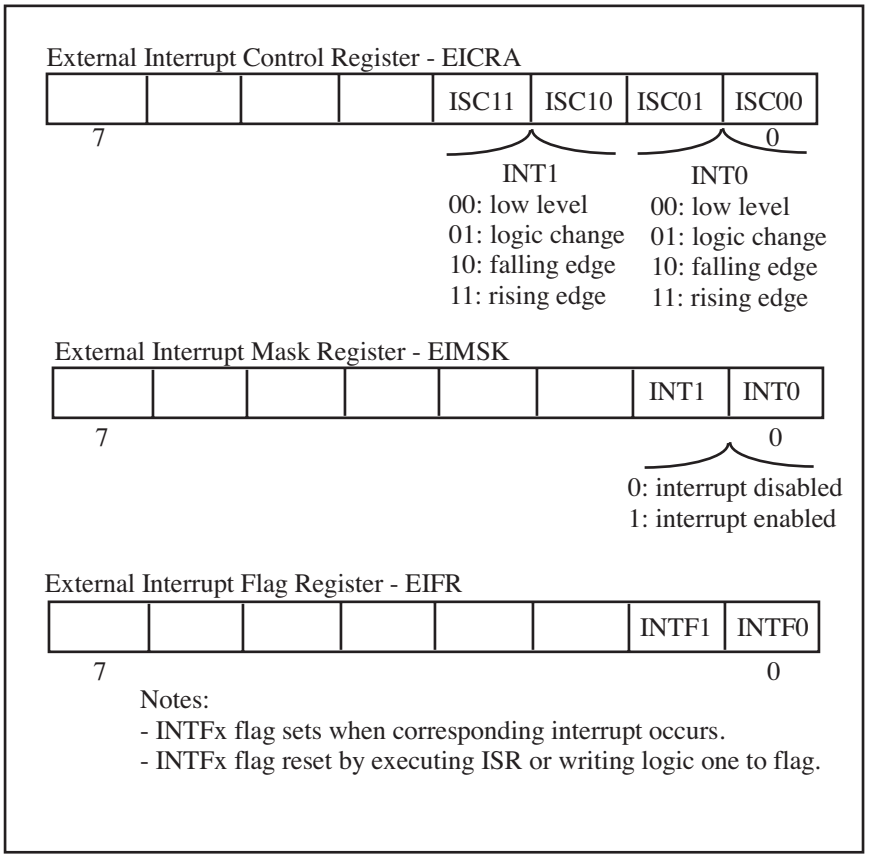


Figure 6.3: ATmega328 Interrupt INT0 and INT1 Registers.

```
:

//*****************************************************************
//function definitions
//*****************************************************************
//initialize_interrupt_int0:  initializes interrupt INT0.
//Note: stack is automatically initialized by the compiler
//*****************************************************************

void initialize_interrupt_int0(void)   //initialize interrupt INT0
{
DDRD = 0xFB;                           //set PD2 (int0) as input
```

```
PORTD &= ~0x04;                          //disable pullup resistor PD2
EIMSK = 0x01;                            //enable INT0
EICRA = 0x03;                            //set positive edge trigger
asm("SEI");                              //global interrupt enable
}

//*****************************************************************
//int0_ISR: interrupt service routine for INT0
//*****************************************************************

void int0_ISR(void)
{

//Insert interrupt specific actions here.

}

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

The INT0 interrupt is reset by executing the associated interrupt service routine or writing a logic one to the INTF0 bit in the External Interrupt Flag Register (EIFR).

6.3.3.2 Programming External Interrupts Using the Arduino Development Environment Built-in Features-Atmega328

The ADE has four built-in functions to support external the INT0 and INT1 external interrupts [www.arduino.cc].

These are the four functions.

- **interrupts()**. This function enables interrupts.
- **noInterrupts()**. This function disables interrupts.
- **attachInterrupt(interrupt, function, mode)**. This function links the interrupt to the appropriate interrupt service routine.
- **detachInterrupt(interrupt)**. This function turns off the specified interrupt.

The Arduino UNO R3 processing board is equipped with two external interrupts: INT0 on DIGITAL pin 2 and INT1 on DIGITAL pin 3. The **attachInterrupt(interrupt, function, mode)** function is used to link the hardware pin to the appropriate interrupt service pin. The three arguments of the function are configured as follows.

- **interrupt.** Interrupt specifies the INT interrupt number: either 0 or 1.

- **function.** Function specifies the name of the interrupt service routine.
- **mode.** Mode specifies what activity on the interrupt pin will initiate the interrupt: **LOW** level on pin, **CHANGE** in pin level, **RISING** edge, or **FALLING** edge.

To illustrate the use of these built-in ADE features, we revisit the previous example.

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

void setup()
{
attachInterrupt(0, int0_ISR, RISING);
}

void loop()
{

//wait for interrupts

}

//*************************************************************
//int0_ISR: interrupt service routine for INT0
//*************************************************************

void int0_ISR(void)
{

//Insert interrupt specific actions here.

}

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

6.3.4 ATMEGA328 INTERNAL INTERRUPT PROGRAMMING

In this example, we use Timer/Counter0 as a representative example on how to program internal interrupts. In the example that follows, we use Timer/Counter0 to provide prescribed delays within our program.

We discuss the ATmega328 timer system in detail in the next chapter. Briefly, the 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.

6.3.4.1 Programming an Internal Interrupt in C-Atmega328-ImageCraft

In this example, the ATmega328 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 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 the function should delay. The function stays within a while loop until the desired number of interrupts has occurred. For example, to delay one second 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 one second 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.

```
//function prototypes*********************************************
                                    //delay number 6.55ms int
void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);//initialize timer0
                                    //overflow interrupt

//interrupt handler definition************************************
                                    //int handler definition
#pragma interrupt_handler timer0_interrupt_isr:17



//global variables************************************************
unsigned int   input_delay;         //counts Timer/Counter0
                                    //Overflow interrupts

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

void main(void)
{
```

```
init_timer0_ovf_interrupt();          //init Timer/Counter0 Overflow

                                      //interrupt - call once at
                                      //beginning of program

:
:
delay(153);                           //1 second delay

}

//*****************************************************************
//int_timer0_ovf_interrupt(): The Timer/Counter0 Overfl. 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 occurs every 6.55ms
TIMSK0 = 0x01;                        //en timer0 overflow interrupt
asm("SEI");                           //enable global interrupt
}

//*****************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared automatically
//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. If a different source frequency is used, the
//clock tick delay value must be recalculated.
//*****************************************************************

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
  }
}

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

6.3.4.2 Programming an Internal Interrupt Using the Arduino Development Environment-Arduino UNO R3

The ADE uses the GNU tool chain and the AVR Libc to compile programs. Internal interrupt configuration uses AVR-GCC conventions. To tie the interrupt event to the correct interrupt service routine, the AVR-GCC interrupt name must be used. These vector names are provided in the right column of Figure 6.2.

In the following sketch, the previous example is configured for use with the ADE using AVR-GCC conventions. Also, the timing functions in the previous example assumed a time base of 10 MHz. The Arduino UNO R3 is clocked with a 16 MHz crystal. Therefore, some of the parameters in the sketch were adjusted to account for this difference in time base.

```
//*****************************************************************
#include <avr/interrupt.h>

unsigned int   input_delay;            //counts Timer/Counter0
                                       //Overflow interrupts


void setup()
```

```
{
init_timer0_ovf_interrupt();         //init Timer/Counter0 Overflow
}

void loop()
{

:

delay(244);                          //1 second delay

:

}

//*****************************************************************
// ISR(TIMER0_OVF_vect) - increments counter on every interrupt.
//*****************************************************************

ISR(TIMER0_OVF_vect)
{
input_delay++;                         //increment overflow counter
}

//*****************************************************************
//int_timer0_ovf_interrupt(): The Timer/Counter0 Overflow interrupt
//is being employed as a time base for a master timer for this
//project. The crystal resonator  operating at 16 MHz is divided by
//256.  The 8-bit Timer0 register (TCNT0) overflows every 256 counts
//or every 4.1 ms.
//*****************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04;                       //div timer0 timebase by 256,
                                     //overflow occurs every 4.1 ms
TIMSK0 = 0x01;                       //en timer0 overflow interrupt
asm("SEI");                          //enable global interrupt
}
```

```
//*****************************************************************
//delay(unsigned int num_of_4_1ms_interrupts): this generic delay
//function provides the specified delay as the number of 4.1 ms
//"clock ticks" from the Timer/Counter0 Overflow interrupt.
//
//Note: this function is only valid when using a 16 MHz crystal or
//ceramic resonator. If a different source frequency is used, the
//clock tick delay value must be recalculated.
//*****************************************************************

void delay(unsigned int number_of_4_1ms_interrupts)
{
TCNT0 = 0x00;                          //reset timer0
input_delay = 0;                       //reset timer0 overflow counter

while(input_delay <= number_of_4_1ms_interrupts)
  {
  ;                                    //wait number of interrupts
  }
}

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

6.4 FOREGROUND AND BACKGROUND PROCESSING

A microcontroller can only process a single instruction at a time. It processes instructions in a fetch-decode-execute sequence as determined by the program and its response to external events. In many cases, a microcontroller has to process multiple events seemingly simultaneously. How is this possible with a single processor?

Normal processing accomplished by the microcontroller is called foreground processing. An interrupt may be used to periodically break into foreground processing, "steal" some clock cycles to accomplish another event called background processing, and then return processor control back to the foreground process.

As an example, a microcontroller controlling access for an electronic door must monitor input commands from a user and generate the appropriate PWM signals to open and close the door. Once the door is in motion, the controller must monitor door motor operation for obstructions, malfunctions, and other safety related parameters. This may be accomplished using interrupts. In this example, the microcontroller is responding to user input status in the fore-

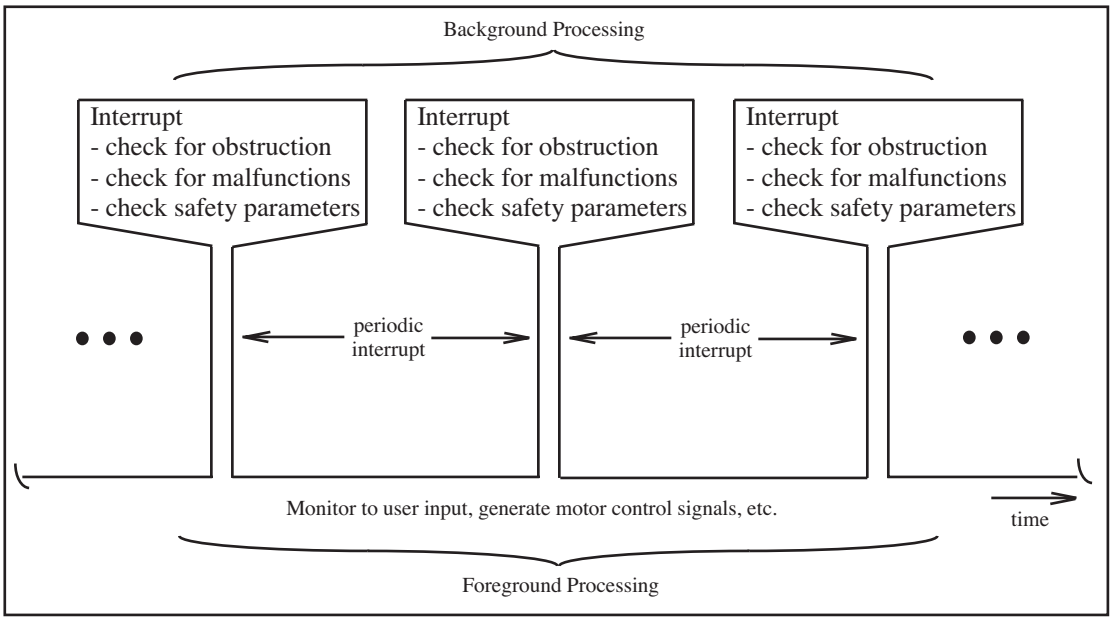


Figure 6.4: Interrupt used for background processing. The microcontroller responds to user input status in the foreground while monitoring safety related status in the background using interrupts.

ground while monitoring safety related status in the background using interrupts as illustrated in Figure 6.4.

Example: This example illustrates foreground and background processing. We use a green LED to indicate when the microcontroller is processing in the foreground and a flashing red LED indicates background processing. A switch is connected to an external interrupt pin (INT0, pin 2). When the switch is depressed, the microcontroller executes the associated interrupt service routine to flash the red LED. The circuit configuration is provided in Figure 6.5.

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

#define green_LED 12
#define red_LED   11
#define ext_sw    2

int switch_value;

void setup()
{
```

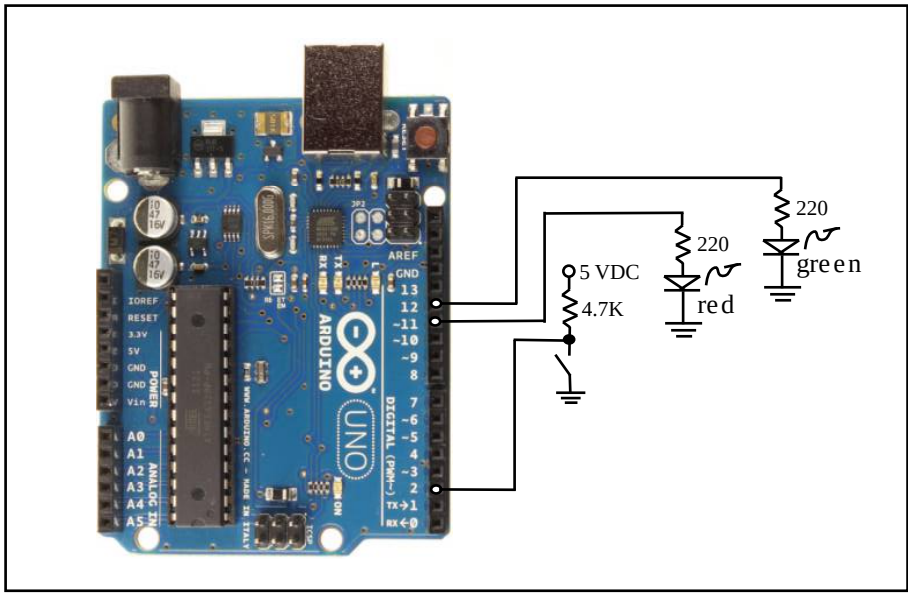


Figure 6.5: Foreground background processing. (UNO R3 illustration used with permission of the Arduino Team (CC BY-NC-SA) [www.arduino.cc].)

```
pinMode(green_LED, OUTPUT);
pinMode(red_LED,   OUTPUT);
pinMode(ext_sw,    INPUT);
                                    //trigger int on INT0 for
                                    //falling edge
attachInterrupt(0, background, FALLING);
}

void loop()
{
digitalWrite(green_LED, HIGH);      //foreground processing
digitalWrite(red_LED,   LOW);
}

void background()                   //background processing
{
unsigned int i;

digitalWrite(green_LED, LOW);
```

```
digitalWrite(red_LED,   HIGH);

for (i=0; i<=64000; i++)             //delay
  {
  asm("nop");
  }

digitalWrite(red_LED,   LOW);

for (i=0; i<=64000; i++)             //delay
  {
  asm("nop");
  }

digitalWrite(red_LED,   HIGH);

for (i=0; i<=64000; i++)             //delay
  {
  asm("nop");
  }

digitalWrite(red_LED,   LOW);

for (i=0; i<=64000; i++)             //delay
  {
  asm("nop");
  }

digitalWrite(red_LED,   HIGH);
}

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

6.5 INTERRUPT EXAMPLES

In this section, we provide several varied examples on using interrupts internal and external to the microcontroller.

6.5.1 EXAMPLE: REAL TIME CLOCK IN C

A microcontroller only "understands" elapsed time in reference to its timebase clock ticks. To keep track of clock time in seconds, minutes, hours etc., a periodic interrupt may be generated for use as a "clock tick" for a real time clock. In this example, we use the Timer 0 overflow to generate a periodic clock tick very 6.55 ms. The ticks are counted in reference to clock time variables and may be displayed on a liquid crystal display. This is also a useful technique for generating very long delays in a microcontroller.

```
//function prototypes*********************************************
                                    //delay number 6.55ms int
void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);//initialize timer0 overflow interrupt


//interrupt handler definition************************************
                                    //int handler definition
#pragma interrupt_handler timer0_interrupt_isr:17


//global variables************************************************
unsigned int days_ctr, hrs_ctr, mins_ctr, sec_ctr, ms_ctr;

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

void main(void)
{
day_ctr = 0; hr_ctr = 0; min_ctr = 0; sec_ctr = 0; ms_ctr = 0;


init_timer0_ovf_interrupt();        //init Timer/Counter0 Overflow

 //interrupt - call once at beginning
                                      //of program
while(1)
  {
  ;                                   //wait for interrupts
  }
}

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

```
//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;                          //div timer0 timebase by 256,
                                        //overflow occurs every 6.55ms

TIMSK0 = 0x01;                          //en 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)
{

//Update millisecond counter
ms_ctr = ms_ctr + 1;                    //increment ms counter

                                        //update second counter
if(ms_ctr == 154)                       //ctr equates 1000 ms at 154
  {
  ms_ctr = 0;                           //reset ms counter
  sec_ctr = sec_ctr + 1;                //increment second counter
  }

//Update minute counter
if(sec_ctr == 60)
  {
  sec_ctr = 0;                          //reset sec counter
```

```
  min_ctr = min_ctr + 1;              //increment min counter
  }

//Update hour counter
if(min_ctr == 60)
  {
  min_ctr = 0;                        //reset min counter
  hr_ctr  = hr_ctr + 1;               //increment hr counter
  }

//Update day counter
if(hr_ctr == 24)
  {
  hr_ctr = 0;                         //reset hr counter
  day_ctr  = day_ctr + 1;             //increment day counter
  }
}

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

6.5.2 EXAMPLE: REAL TIME CLOCK USING THE ARDUINO DEVELOPMENT ENVIRONMENT

In this example, we reconfigure the previous example using the ADE. The timing functions in the previous example assumed a time base of 10 MHz. The Arduino UNO R3 is clocked with a 16 MHz crystal. Therefore, some of the parameters in the sketch are adjusted to account for this difference in time base.

```
//*************************************************************
#include <avr/interrupt.h>

//global variables*****************************************************
unsigned int days_ctr, hrs_ctr, mins_ctr, sec_ctr, ms_ctr;

void setup()
{
day_ctr = 0; hr_ctr = 0; min_ctr = 0; sec_ctr = 0; ms_ctr = 0;
init_timer0_ovf_interrupt();          //init. Timer/Counter0 Overflow
}

void loop()
```

```
{
:
:                                   //wait for interrupts
:
}

//*****************************************************************
// ISR(TIMER0_OVF_vect) Timer0 interrupt service routine.
//
//Note: Timer overflow 0 is cleared by hardware when executing the
//corresponding interrupt handling vector.
//*****************************************************************

ISR(TIMER0_OVF_vect)
{
                                    //Update millisecond counter
ms_ctr = ms_ctr + 1;                //increment ms counter

                                    //Update second counter
                                    //ctr equates 1000 ms at 244

if(ms_ctr == 244)                   //each clock tick is 4.1 ms
  {
  ms_ctr = 0;                       //reset ms counter
  sec_ctr = sec_ctr + 1;            //increment second counter
  }

//Update minute counter
if(sec_ctr == 60)
  {
  sec_ctr = 0;                      //reset sec counter
  min_ctr = min_ctr + 1;            //increment min counter
  }

//Update hour counter
if(min_ctr == 60)
  {
  min_ctr = 0;                      //reset min counter
  hr_ctr  = hr_ctr + 1;             //increment hr counter
```

```
  }

//Update day counter
if(hr_ctr == 24)
  {
  hr_ctr = 0;                          //reset hr counter
  day_ctr  = day_ctr + 1;              //increment day counter
  }
}

//*****************************************************************
//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 16 MHz
//is divided by 256.  The 8-bit Timer0 register (TCNT0) overflows
//every 256 counts or every 4.1 ms.
//*****************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04;                         //divide timer0 timebase by 256,
                                       //overflow occurs every 4.1 ms
TIMSK0 = 0x01;                         //en timer0 overflow interrupt
asm("SEI");                            //enable global interrupt
}
//*****************************************************************
```

6.5.3 EXAMPLE: INTERRUPT DRIVEN USART IN C

In Chapter 5, we discussed the serial communication capability of the USART in some detail. In the following example, we revisit the USART and use it in an interrupt driven mode.

Example. You have been asked to evaluate a new positional encoder technology. The encoder provides 12-bit resolution. The position data is sent serially at 9600 Baud as two sequential bytes as shown in Figure 6.6. The actual encoder is new technology and production models are not available for evaluation.

Since the actual encoder is not available for evaluation, another Microchip ATmega328 will be used to send signals in identical format and Baud rate as the encoder. The test configuration is illustrated in Figure 6.7. The ATmega328 on the bottom serves as the positional encoder. The microcontroller is equipped with two pushbuttons at PD2 and PD3. The pushbutton at PD2 provides a debounced input to open a simulated door and increment the positional encoder. The

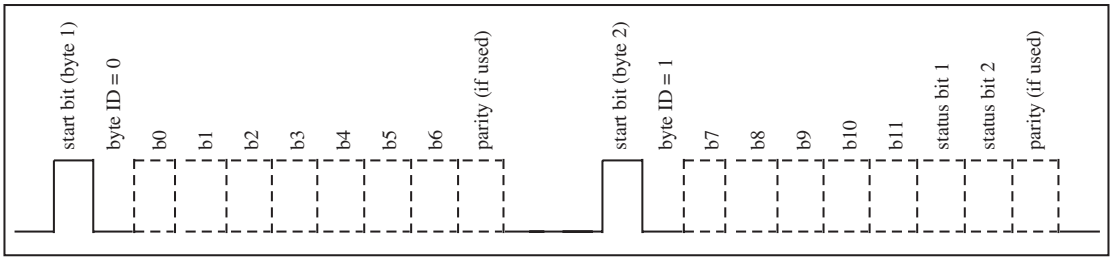


Figure 6.6: Encoder data format. The position data is sent serially at 9600 Baud as two sequential bytes.

pushbutton at PD3 provides a debounced input to close a simulated door and decrement the positional encoder. The current count of the encoder (eight most significant bits) is fed to a digital-to-analog converter (DAC0808) to provide an analog representation.

The positional data from the encoder is sent out via the USART in the format described in Figure 6.6. The top ATmega328 receives the positional data using interrupt driven USART techniques. The current position is converted to an analog signal via the DAC. The transmitted and received signals may be compared at the respective DAC outputs.

Provided below is the code for the ATmega328 that serves as the encoder simulator followed by the code for receiving the data.

```
//*************************************************************
//author: Steven Barrett, Ph.D., P.E.
//last revised: March 23, 2020
//file: encode.c
//target controller: MICROCHIP ATmega328
//
//ATmega328 clock source: internal 8 MHz clock
//
//MICROCHIP AVR ATmega328PV Controller Pin Assignments
//Chip Port Function I/O Source/Dest Asserted Notes
//
//Pin  1 to system reset circuitry
//Pin  2 PD0: USART receive pin (RXD)
//Pin  3 PD1: USART transmit pin (TXD)
//Pin  4 PD2 to active high RC debounced switch - Open
//Pin  5 PD3 to active high RC debounced switch - Close
//Pin  7 Vcc - 1.0 uF to ground
//Pin  8 Gnd
//Pin  9 PB6 to pin A7(11) on DAC0808
```

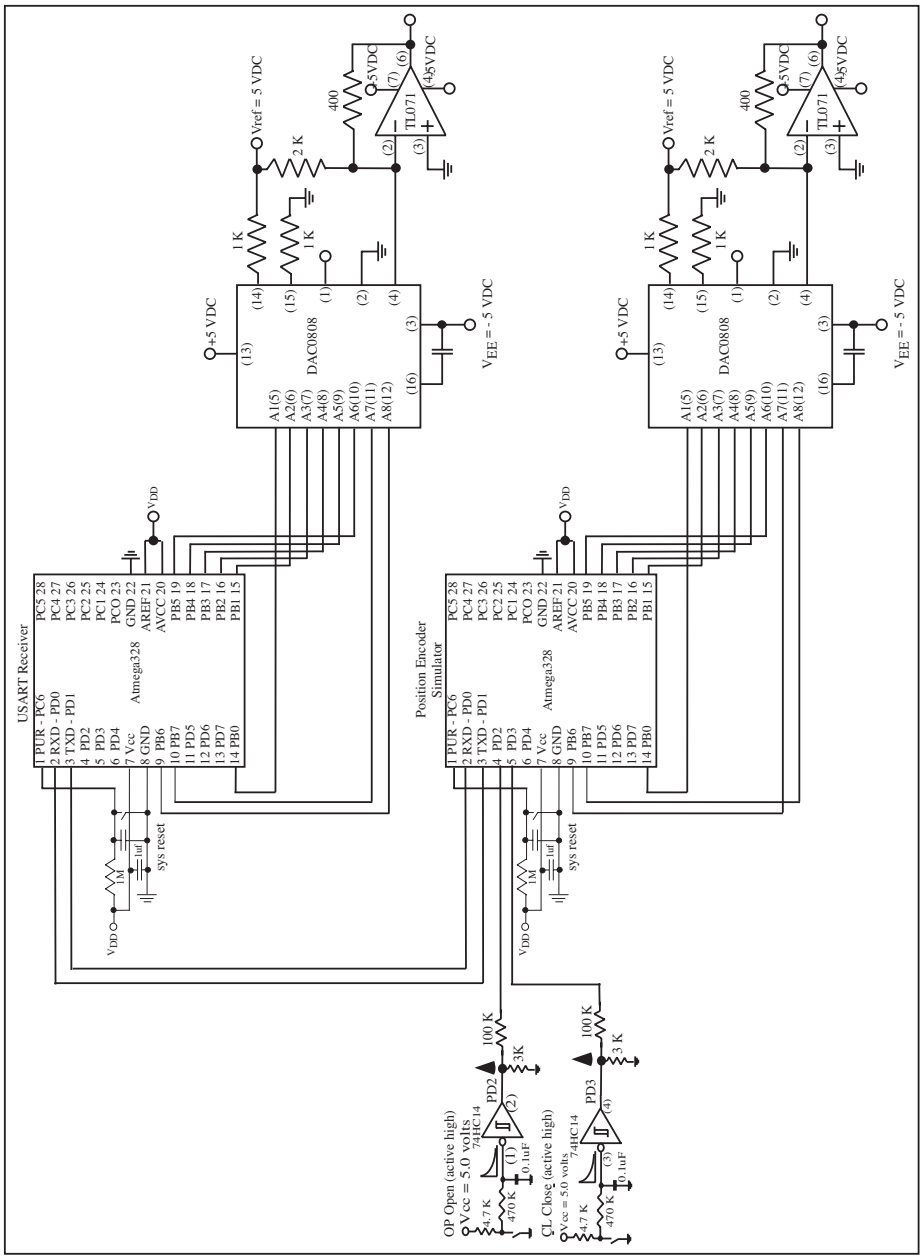


Figure 6.7: Encoder test configuration.

```
//Pin 10 PB7 to pin A8(12) on DAC0808
//Pin 14 PB0 to pin A1(5)  on DAC0808
//Pin 15 PB1 to pin A2(6)  on DAC0808
//Pin 16 PB2 to pin A3(7)  on DAC0808
//Pin 17 PB3 to pin A4(8)  on DAC0808
//Pin 18 PB4 to pin A5(9)  on DAC0808
//Pin 19 PB5 to pin A6(10) on DAC0808
//Pin 20 AVCC to 5 VDC
//Pin 21 AREF to 5 VDC
//Pin 22 Ground
//*****************************************************************

//include files*****************************************************
#include<iom328v.h>
#include<macros.h>

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

 //delay specified number 6.55ms int
void initialize_ports(void);           //initializes ports
void USART_init(void);
void USART_TX(unsigned char data);

//main program******************************************************
//global variables
unsigned char   old_PORTD = 0x08;     //present value of PORTD
unsigned char   new_PORTD;            //new values of PORTD
unsigned int    door_position = 0;

void main(void)
{
initialize_ports();                   //return LED config to default
USART_init();

//main activity loop - checks PORTD to see if either PD2 (open)
//or PD3 (close) was depressed.
//If either was depressed the program responds.
```

```
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)
{
//PORTB
DDRB=0xff;                            //PORTB[7-0] output
PORTB=0x00;                           //initialize low

//PORTC
DDRC=0xff;                            //set PORTC[7-0] as output
PORTC=0x00;                           //initialize low

//PORTD
DDRD=0xf2;                            //set PORTD[7-4, 0] as output
PORTD=0x00;                           //initialize low
}

//*****************************************************************
//read_new_input: functions polls PORTD for a change in status. If
//status change has occurred, appropriate function for status change
//is called
//Pin  4 PD2 to active high RC debounced switch - Open
//Pin  5 PD3 to active high RC debounced switch - Close
//*****************************************************************

void read_new_input(void)
{
unsigned int   gate_position;         //measure instantaneous position
```

```
                                        //of gate
unsigned int i;
unsigned char ms_door_position, ls_door_position, DAC_data;

new_PORTD = (PIND & 0x0c);
                                       //mask all pins but PORTD[3:2]
if(new_PORTD != old_PORTD){
  switch(new_PORTD){                   //process change in PORTD input

    case 0x01:                         //Open
      while(PIND == 0x04)
        {
        //split into two bytes
        ms_door_position=(unsigned char)(((door_position >> 6)
                                          &(0x00FF))|0x01);
        ls_door_position=(unsigned char)(((door_position << 1)
                                          &(0x00FF))&0xFE);
        //TX data to USART
        USART_TX(ms_door_position);
        USART_TX(ls_door_position);

        //format data for DAC and send to DAC on PORTB
        DAC_data=(unsigned char)((door_position >> 4)&(0x00FF));
        PORTB = DAC_data;

        //increment position counter
        if(door_position >= 4095)
           door_position = 4095;
        else
          door_position++;
        }
      break;

    case 0x02:                         //Close
      while(PIND == 0x02)
        {
                                       //split into two bytes
        ms_door_position=(unsigned char)(((door_position >>6)
                                          &(0x00FF))|0x01);
```

```
        ls_door_position=(unsigned char)(((door_position <<1)
                                        &(0x00FF))&0xFE);
                                    //TX data to USART
        USART_TX(ms_door_position);
        USART_TX(ls_door_position);

                                    //format data for DAC
                                    //and send to DAC on PORTB
        DAC_data=(unsigned char)((door_position >> 4)&(0x00FF));

        PORTB = DAC_data;

                                    //decrement position counter
        if(door_position <= 0)
          door_position = 0;
        else
          door_position-;
        }
        break;

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

//*****************************************************************
//USART_init: initializes the USART system
//
//Note: ATmega328 clocked by internal 8 MHz clock
//*****************************************************************

void USART_init(void)
{
UCSRA = 0x00;               //control
register initialization
UCSRB = 0x08;               //enable transmitter
UCSRC = 0x86;               //async, no parity,
                            //1 stop bit, 8 data bits
```

```
                              //Baud Rate initialization
                              //8 MHz clock requires UBRR value of 51
                              // or 0x0033 to achieve 9600 Baud rate
UBRRH = 0x00;
UBRRL = 0x33;
}

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

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

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

Receive ATmega328 code follows.

```
//*****************************************************************
//author: Steven Barrett, Ph.D., P.E.
//last revised: March 23, 2020
//file: receive.c
//target controller: MICROCHIP ATmega328
//
//ATmega328 clock source: internal 8 MHz clock
//
//MICROCHIP AVR ATmega328PV Controller Pin Assignments
//Chip Port Function I/O Source/Dest Asserted Notes
//
//Pin  1 to system reset circuitry
//Pin  2 PD0: USART receive pin (RXD)
//Pin  3 PD1: USART transmit pin (TXD)
//Pin  4 PD2 to active high RC debounced switch - Open
//Pin  5 PD3 to active high RC debounced switch - Close
//Pin  7 Vcc - 1.0 uF to ground
```

```
//Pin  8 Gnd
//Pin  9 PB6 to pin A7(11) on DAC0808
//Pin 10 PB7 to pin A8(12) on DAC0808
//Pin 14 PB0 to pin A1(5)  on DAC0808
//Pin 15 PB1 to pin A2(6)  on DAC0808
//Pin 16 PB2 to pin A3(7)  on DAC0808
//Pin 17 PB3 to pin A4(8)  on DAC0808
//Pin 18 PB4 to pin A5(9)  on DAC0808
//Pin 19 PB5 to pin A6(10) on DAC0808
//Pin 20 AVCC to 5 VDC
//Pin 21 AREF to 5 VDC
//Pin 22 Ground
//*************************************************************

//include files*********************************************
#include<iom328v.h>
#include<macros.h>
#include<eeprom.h>                    //EEPROM support functions

#pragma data: eeprom
unsigned int door_position_EEPROM
#pragma data:data


//function prototypes***************************************
void initialize_ports(void);          //initializes ports
void InitUSART(void);
unsigned char USART_RX(void);
                                      //interrupt handler def
#pragma interrupt_handler USART_RX_interrupt_isr: 19

//main program**********************************************
unsigned int    door_position = 0;
unsigned char   data_rx;
unsigned int    dummy1 = 0x1234;
unsigned int    keep_going =1;
unsigned int    loop_counter = 0;
unsigned int    ms_position, ls_position;
```

```
void main(void)
{
initialize_ports();                  //return LED config to default
USART_init();
                                     //limited startup features
                                     //main activity loop
                                     //continually cycle thru loop
                                     //waiting for USART data

while(1)
  {
                                     //continuous loop waiting for
                                     //interrupts

  _StackCheck();                     //check for stack overflow
  }
}//end main

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

void initialize_ports(void)
{
//PORTB
DDRB=0xff;                           //PORTB[7-0] output
PORTB=0x00;                          //initialize low

//PORTC
DDRC=0xff;                           //set PORTC[7-0] as output
PORTC=0x00;                          //initialize low

//PORTD
DDRD=0xff;                           //set PORTD[7-0] as output
PORTD=0x00;                          //initialize low
}

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

```
//USART_init: initializes the USART system
//
//Note: ATmega328 clocked by internal 8 MHz clock
//*****************************************************************

void USART_init(void)
{
UCSRA = 0x00;                 //control
register initialization
UCSRB = 0x08;                 //enable transmitter
UCSRC = 0x86;                 //async, no parity, 1 stop bit, 8 data
                              //Baud Rate initialization
                              //8 MHz clock requires UBRR value of 51
                              // or 0x0033 to achieve 9600 Baud rate
UBRRH = 0x00;
UBRRL = 0x33;
}

//*****************************************************************
//USART_RX_interrupt_isr
//*****************************************************************

void USART_RX_interrupt_isr(void)
{
unsigned char data_rx, DAC_data;
unsigned int  ls_position, ms_position;

//Receive USART data
data_rx = UDR;

//Retrieve door position data from EEPROM
EEPROM_READ((int) &door_position_EEPROM, door_position);

//Determine which byte to update
if((data_rx & 0x01)==0x01)                    //Byte ID = 1
  {
  ms_position = data_rx;

  //Update bit 7
```

```
  if((ms_position & 0x0020)==0x0020)           //Test for logic 1
    door_position = door_position | 0x0080;    //Set bit 7 to 1
  else
    door_position = door_position & 0xff7f;    //Reset bit 7 to 0

  //Update remaining bits
  ms_position = ((ms_position<<6) & 0x0f00);
  //shift left 6-blank other bits
  door_position = door_position & 0x00ff;      //Blank ms byte
  door_position = door_position | ms_position; //Update ms byte
  }
else                                           //Byte ID = 0
  {
  ls_position = data_rx;                       //Update ls_position
                                               //Shift right 1-blank
  ls_position = ((ls_position >> 1) & 0x007f); //other bits

  if((door_position & 0x0080)==0x0080)
        //Test bit 7 of curr position
    ls_position = ls_position | 0x0080;        //Set bit 7
  else
    ls_position = ls_position & 0xff7f;        //Reset bit 7 to 0
  door_position = door_position & 0xff00;      //Blank ls byte
  door_position = door_position | ls_position; //Update ls byte
  }

//Store door position data to EEPROM
EEPROM_WRITE((int) &door_position_EEPROM, door_position);


//format data for DAC and send to DAC on PORT C
DAC_data=(unsigned char)((door_position >> 4)&(0x00FF));

PORTB= DAC_data;
}

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

6.6 SUMMARY

In this chapter, we provided an introduction to the interrupt features available aboard the ATmega328 and the Arduino UNO R3 processing board. We also discussed how to program an interrupt for proper operation and provided representative samples for an external interrupt and an internal interrupt.

6.7 REFERENCES

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

[2] Arduino homepage. www.arduino.cc

[3] *Microchip ATmega328 PB AVR Microcontroller with Core Independent Peripherals and Pico Power Technology DS40001906C*, Microchip Technology Incorporation, 2018. www.microchip.com

6.8 CHAPTER PROBLEMS

1. What is the purpose of an interrupt?
2. Describe the flow of events when an interrupt occurs.
3. Describe the interrupt features available with the ATmega328.
4. Describe the built-in interrupt features available with the Arduino Development Environment.
5. What is the interrupt priority? How is it determined?
6. What steps are required by the system designer to properly configure an interrupt?
7. How is the interrupt system turned "ON" and "OFF"?
8. A 10-MHz ceramic resonator is not available. Redo the example of the Timer/Counter0 Overflow interrupt provided with a timebase 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 modification? How could the function be modified for longer delays?
10. In the text, we provided a 24-h timer (hh:mm:ss:ms) using the Timer/Counter0 Overflow interrupt. What is the accuracy of the timer? How can it be improved?
11. Adapt the 24-h timer example to generate an active high logic pulse on a microcontroller pin of your choice for 3 s. The pin should go logic high three weeks from now.

12. What are the concerns when using multiple interrupts in a given application?
13. How much time can background processing relative to foreground processing be implemented?
14. What is the advantage of using interrupts over polling techniques?
15. Can the USART transmit and interrupt receive system provided in the chapter be adapted to operate in the Arduino Development Environment? Explain in detail.

CHAPTER 7

Embedded Systems Design

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

- define an embedded system;
- list all aspects related to the design of an embedded system;
- provide a step-by-step approach to embedded system design;
- discuss design tools and practices related to embedded systems design; and
- apply embedded system design practices in the design of a microcontroller system employing several interacting subsystems.

7.1 OVERVIEW

In this chapter,[1] we begin with a definition of just what is 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 examples.

7.2 WHAT IS AN EMBEDDED SYSTEM?

An embedded system contains a microcontroller to accomplish 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 [1]]. Through careful application of this material you will be able to design and prototype embedded systems based on the ATmega328 microcontroller.

7.3 EMBEDDED SYSTEM DESIGN PROCESS

In this section, we provide 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

[1]The information on embedded system design first appeared in *Microcontroller Fundamentals for Engineers and Scientists*, Morgan & Claypool Publishers, 2006. It has been adapted with permission for the Microchip ATmega328.

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 7.1 using a Unified Modeling Language (UML) activity diagram. We discuss the UML activity diagrams later in the chapter.

7.3.1 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 what the system is supposed to do. 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, the designer interacts 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.

7.3.2 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.

7.3.3 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 speaking, a hardware only solution executes

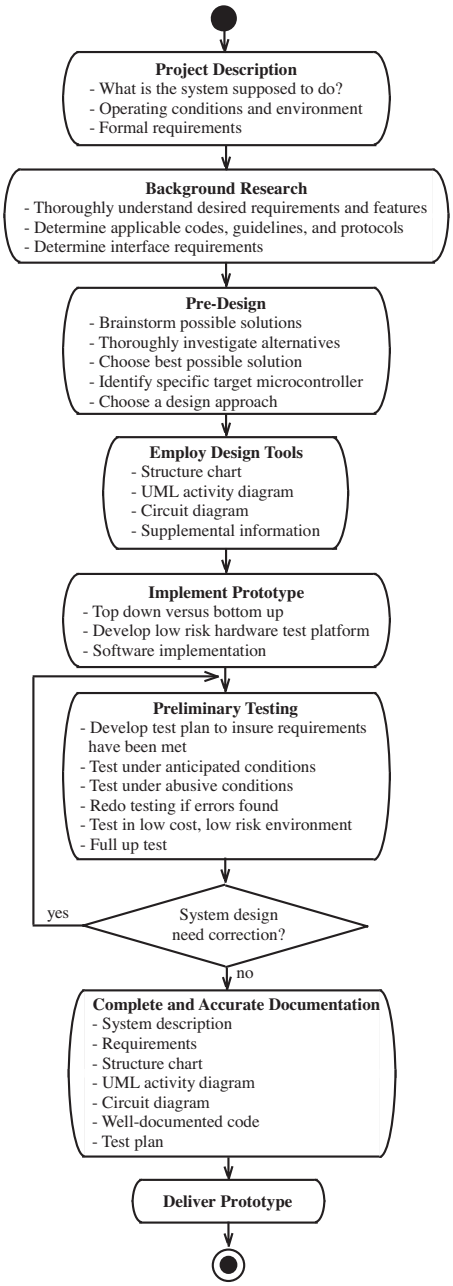


Figure 7.1: Embedded system design process.

faster; 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?

7.3.4 DESIGN

With a clear view of system requirements and features, a general partition determined between hardware and software, and a specific microcontroller chosen, 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:

- 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 (Barrett and Pack [2]).

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

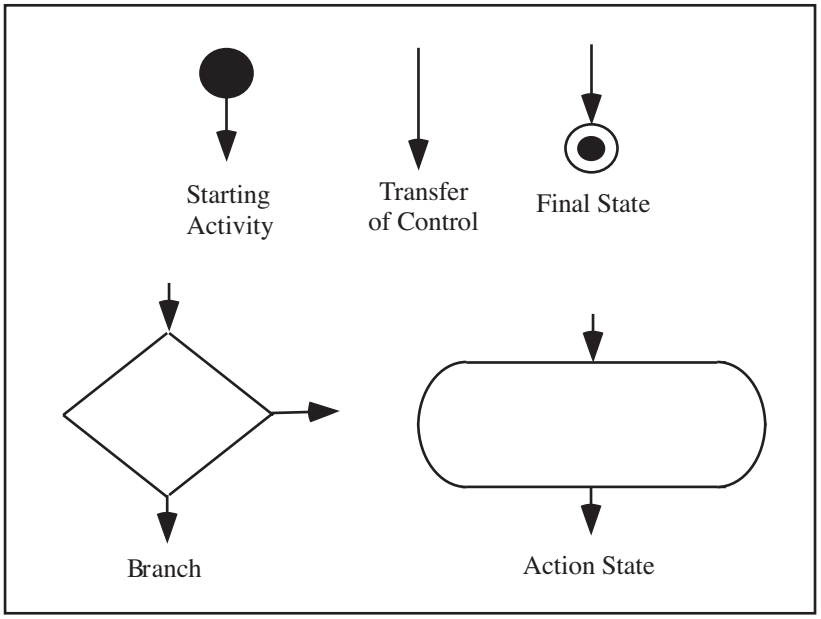


Figure 7.2: UML activity diagram symbols. Adapted from Fowler and Scott [5].

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: 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 7.2 (Fowler and Scott [5]).

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 [6]).

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.

7.3.5 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 insure the software is properly operating before using it to control the actual components.

7.3.6 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!

7.3.7 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. For the remainder of the chapter, we employ these design techniques in several examples.

7.4 EXAMPLE: AUTOMATED FAN COOLING SYSTEM

In this example we describe an embedded system application to control the temperature of a room or some device. The system is illustrated in Figure 7.3. An LM34 precision Fahrenheit temperature sensor (PORTC[0]) is used to monitor the instantaneous temperature of the room or device of interest. The current temperature is displayed on the Liquid Crystal Display (LCD). The LCD (AND671GST) is parallel configured. Detailed LCD support functions are provided in the example.

We send a 1-KHz PWM signal to a cooling fan (M) whose duty cycle is set from 50–90% using the potentiometer connected to PORTC[2]. The PWM signal should last until the temperature of the LM34 cools to a value as set by another potentiometer (PORTC[1]). The LM34 provides 10 mV per degree Fahrenheit. When the temperature of the LM34 falls below the potentiometer set level, the cooling fan is shut off. If the temperature falls while the fan is active, the PWM signal should gently return to zero, and wait for further temperature changes.

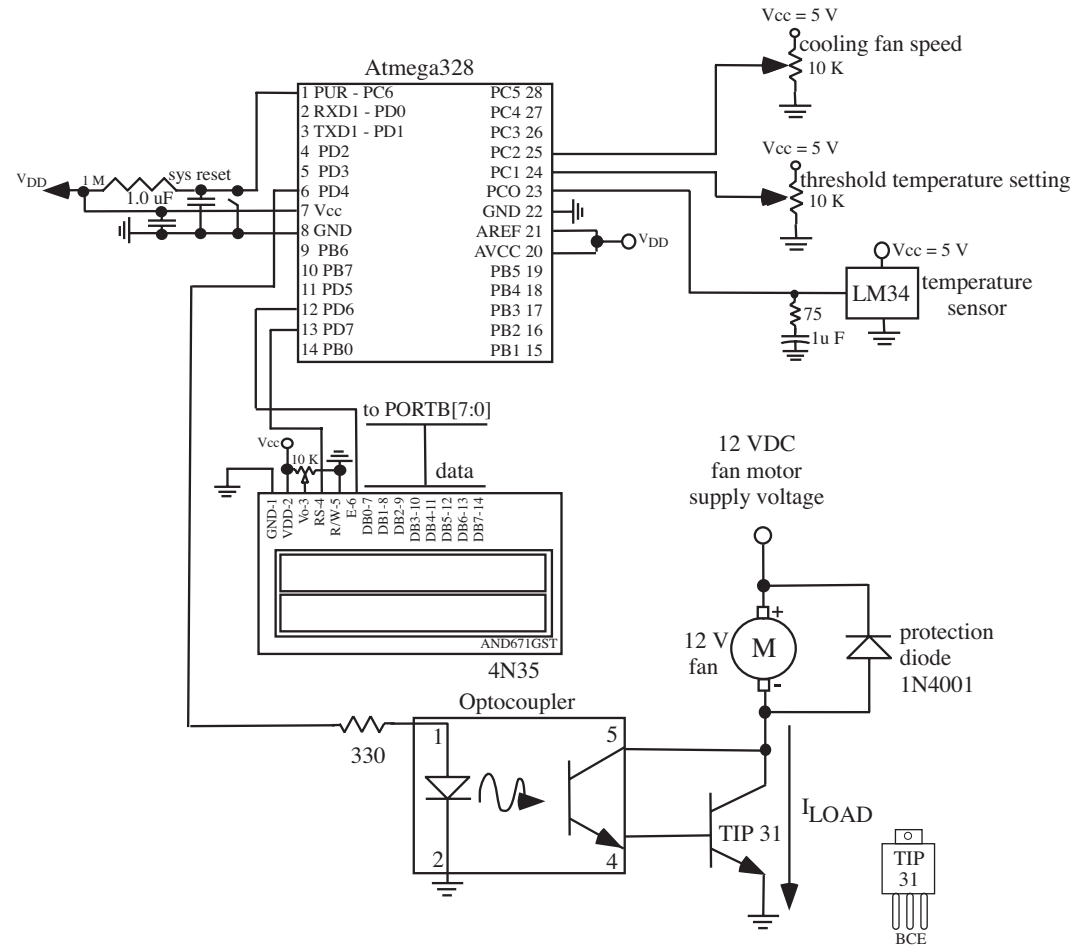


Figure 7.3: Automated fan cooling system.

As an example, to set the desired temperature threshold to 70°F, the potentiometer should be set to 0.7 VDC.

Provided below is the embedded code for the system. This solution was developed by Geoff Luke, UW MSEE, as a laboratory assignment for an Industrial Control class.

```
//*****************************************************************
//Geoff Luke
//EE 5880 - Industrial Controls
//PWM Fan Control
```

```
//Last Updated: April 13, 2020
//*************************************************************
//Description: This program reads the voltage from an LM34 temp
//sensor then sends the corresponding temperature to an LCD.
//If the sensed temperature is greater than the temperature set
//by a potentiometer, then a PWM signal is turned on to trigger a
//DC fan with duty cycle set by another potentiometer.
//
//The ATmega328 is clocked from 4 MHz ceramic resonator
//
//Ports:
//  PORTB[7:0]: data output to LCD
//  PORTD[7:6]: LCD control pins
//  PORTC[0]:   LM34 temperature sensor
//  PORTC[1]:   threshold temperature 10 mV per degree F
//  PORTC[2]:   fan speed
//  PORTD[4]:   PWM channel B output
//
//*************************************************************

//include files*********************************************
#include<iom328pv.h>

//function prototypes***************************************
void initializePorts(void);
void initializeADC(void);
unsigned int readADC(unsigned char);
void LCD_init(void);
void putChar(unsigned char);
void putcommand(unsigned char);
void voltageToLCD(unsigned int);
void temperatureToLCD(unsigned int);
void PWM(unsigned int);
void delay(void);

int main(void)
{
unsigned int tempVoltage, tempThreshold;
```

```
initializePorts();
initializeADC();
LCD_init();

while(1)
  {
  tempVoltage = readADC(0);
  temperatureToLCD(tempVoltage);
  tempThreshold = readADC(1);
  if(tempVoltage > tempThreshold)
    {
    PWM(1);
    while(tempVoltage > tempThreshold)
      {
      tempVoltage = readADC(0);
      temperatureToLCD(tempVoltage);
      tempThreshold = readADC(1);
      }
    OCR1BL = 0x00;
    }
   }
return 0;
}

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

void initializePorts(void)
{
DDRD = 0xFF;
DDRC = 0xFF;
DDRB = 0xFF;
}

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

void initializeADC(void)
{
//select channel 0
ADMUX = 0;
```

```
//enable ADC and set module enable ADC and
//set module prescalar to 8
ADCSRA = 0xC3;

//Wait until conversion is ready
while(!(ADCSRA & 0x10));

//Clear conversion ready flag

ADCSRA |= 0x10;
}

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

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

ADMUX = channel;                    //Select channel
ADCSRA |= 0x43;                     //Start conversion

                                    //ADC module prescalar to 8
                                    //critical accurate ADC results
while (!(ADCSRA & 0x10));           //Check if conversion is ready
ADCSRA |= 0x10;                     //Clear conv rdy flag-set 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
}

//*****************************************************************
//LCD_Init: initialization 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: PORTB
//LCD RS (pin 4) MICROCHIP ATmega16: PORTD[7]
//LCD E (pin 6) MICROCHIP ATmega16: PORTD[6]
//*****************************************************************

void LCD_init(void)
{
delay();
delay();
delay();
                                    // output command string to
                                    //initialize LCD
putcommand(0x38);                   //function set 8-bit
delay();
putcommand(0x38);                   //function set 8-bit
delay();
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
}

//*****************************************************************
//putchar: prints specified ASCII character to LCD
//*****************************************************************

void putChar(unsigned char c)
{
DDRB = 0xff;                        //set PORTB as output
DDRD = DDRD|0xC0;                   //make PORTD[7:6] output
PORTB = c;
PORTD = PORTD|0x80;                 //RS=1
PORTD = PORTD|0x40;                 //E=1
PORTD = PORTD&0xbf;                 //E=0
```

```
delay();
}

//*****************************************************************
//performs specified LCD related command
//*****************************************************************

void putcommand(unsigned char d)
{
DDRB = 0xff;                          //set PORTB as output
DDRD = DDRD|0xC0;                     //make PORTD[7:6] output
PORTD = PORTD&0x7f;                   //RS=0
PORTB = d;
PORTD = PORTD|0x40;                   //E=1
PORTD = PORTD&0xbf;                   //E=0
delay();
}


//*****************************************************************
//delay
//Clock source:  4 MHz ceramic resonator
//Delay: 5 ms
//*****************************************************************

void delay(void)
{
unsigned int i;

for(i=0; i<20000; i++)
  {
  asm("nop");
  }
}

//*****************************************************************
//This function transforms the ADC voltage measured as an unsigned
//int, isolates each voltage digit, converts to an ASCII equivalent,
//and displays the result on the LCD.
```

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

void voltageToLCD(unsigned int ADCValue)
{
float voltage;
unsigned int ones, tenths, hundredths;

voltage = (float)ADCValue*5.0/1024.0;

ones = (unsigned int)voltage;
tenths = (unsigned int)((voltage-(float)ones)*10);
hundredths = (unsigned int)(((voltage-(float)ones)*10
              -(float)tenths)*10);

putcommand(0x80);

putChar((unsigned char)(ones)+48);
putChar('.');
putChar((unsigned char)(tenths)+48);
putChar((unsigned char)(hundredths)+48);
putChar('V');
putcommand(0xC0);
}

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

void temperatureToLCD(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(0x80);
putChar((unsigned char)(tens)+48);
putChar((unsigned char)(ones)+48);
putChar('.');
putChar((unsigned char)(tenths)+48);
putChar('F');
}

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

void PWM(unsigned int PWM_incr)
{

unsigned int fan_Speed_int;
float fan_Speed_float;
int PWM_duty_cycle;

fan_Speed_int = readADC(0x02);       //fan speed setting

//unsigned int convert to max duty cycle setting:
//  0 VDC =  50
//  5 VDC = 100

fan_Speed_float = ((float)(fan_Speed_int)/(float)(0x0400));

//convert volt to PWM constant 127-255
fan_Speed_int = (unsigned int)((fan_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: 980 Hz
                                        //Initiate PWM duty cycle
                                        //variables
PWM_duty_cycle = 0;
```

```
OCR1BH = 0x00;
OCR1BL = (unsigned char)(PWM_duty_cycle);//PWM duty cycle Ch B to 0
                                          //Ramp to fan Speed in 1.6s
OCR1BL = (unsigned char)(PWM_duty_cycle);//set PWM duty cycle Ch B

while (PWM_duty_cycle < fan_Speed_int)
  {
  if(PWM_duty_cycle < fan_Speed_int)     //increment duty cycle
  PWM_duty_cycle=PWM_duty_cycle + PWM_incr;
                                          //set PWM duty cycle Ch B
  OCR1BL = (unsigned char)(PWM_duty_cycle);
  }
}

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

7.5 AUTONOMOUS MAZE NAVIGATING ROBOT

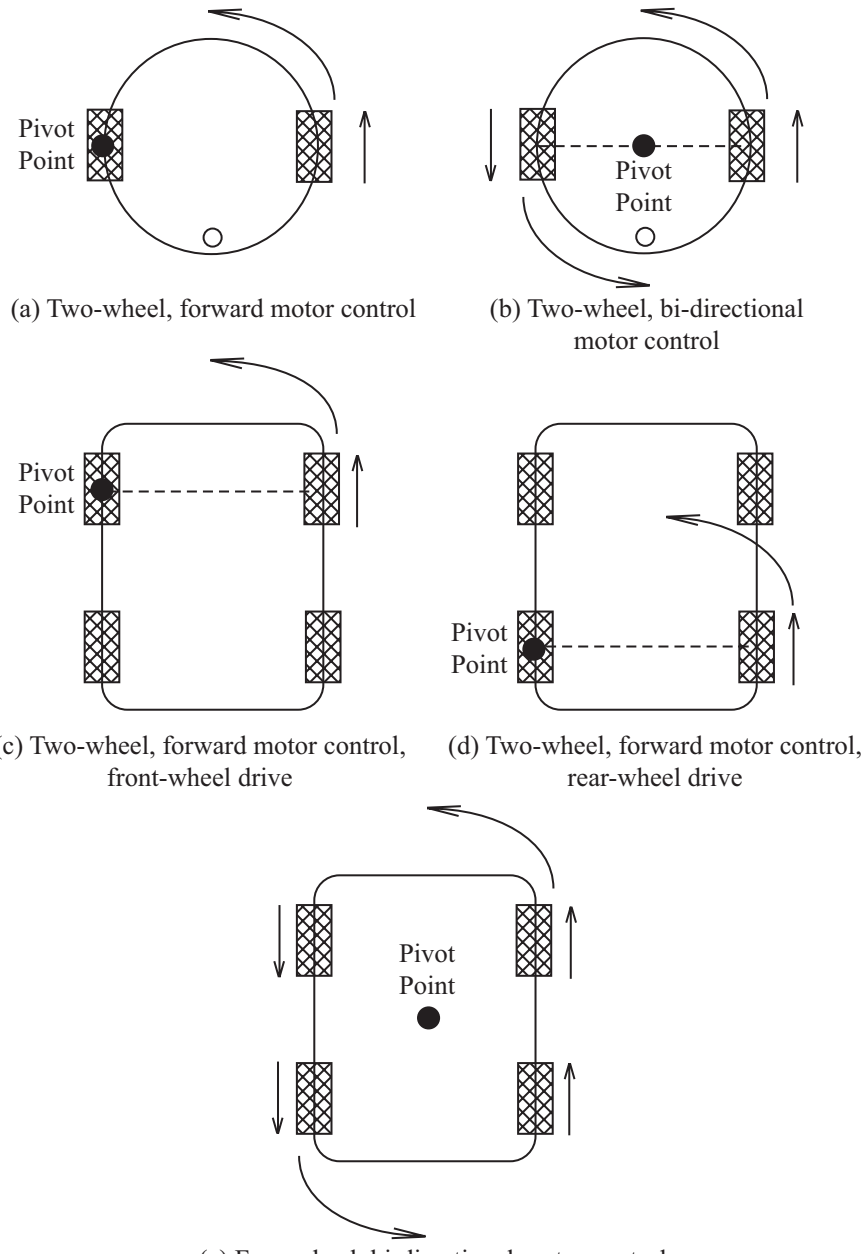
In this example we investigate an autonomous navigating robot design. Before delving into the design, it is helpful to review the fundamentals of robot steering and motor control. Figure 7.4 illustrates the fundamental concepts. Robot steering is dependent upon the number of powered wheels and whether the wheels are equipped with unidirectional or bidirectional control. Recall from *Arduino I: Getting Started* an H-bridge is typically required for bidirectional control of a DC motor. Additional robot steering configurations are possible.

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 7.5. 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.

7.5.1 DAGU ROVER 5 TRACKED ROBOT

We equip the Dagu Rover 5 Tracked Robot for control by ATmega328 as a maze navigating robot. The Rover 5 kit is available from Jameco Electronics (www.jameco.com). The Rover 5 kits comes in three variants:

- ROV5 – 1 is equipped with two motor and no motor encoders.
- ROV5 – 2 is equipped with two motor and two motor encoders.



(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 7.4: Robot control configurations.

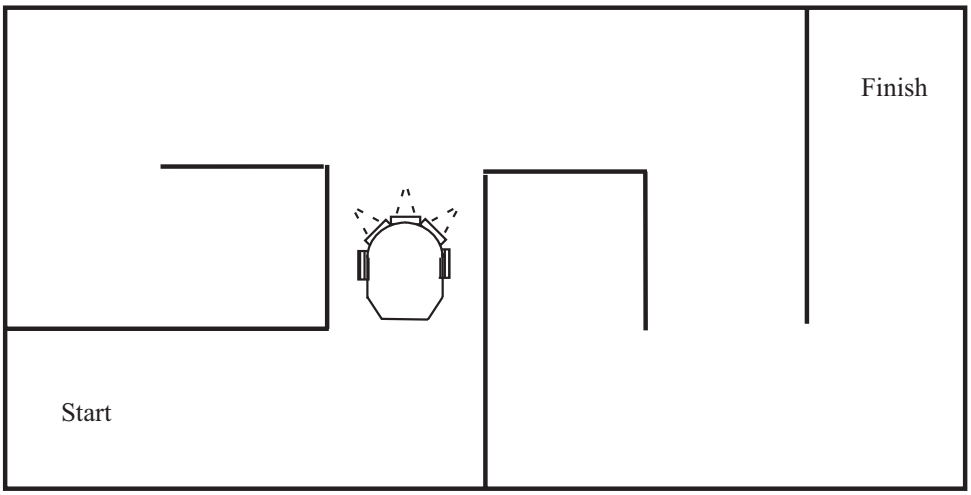


Figure 7.5: Autonomous robot within maze.

- ROV5 – 3 is equipped with four motor and four motor encoders.

In this example, we use the ROV5-1 platform. It is controlled by two 7.2 VDC motors which independently drive a left and right wheel.

We equip the Rover 5 platform with three Sharp GP2Y0A21YKOF infrared (IR) sensors as shown in Figure 7.7. The sensors are available from SparkFun Electronics (www.sparkfun.com). We mount the sensors on a bracket constructed from thin aluminum. The bracket is attached to the Pololu RP5/Rover 5 extension plate (#1530) [www.pololu.com].

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 sensor are provided in Figure 7.6. 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 ATmega328.

7.5.2 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. 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 7.12 and the actions specified in the robot action truth table (Figure 7.8) and transform both into a coded algorithm. This may seem formidable but we take it a step at a time.

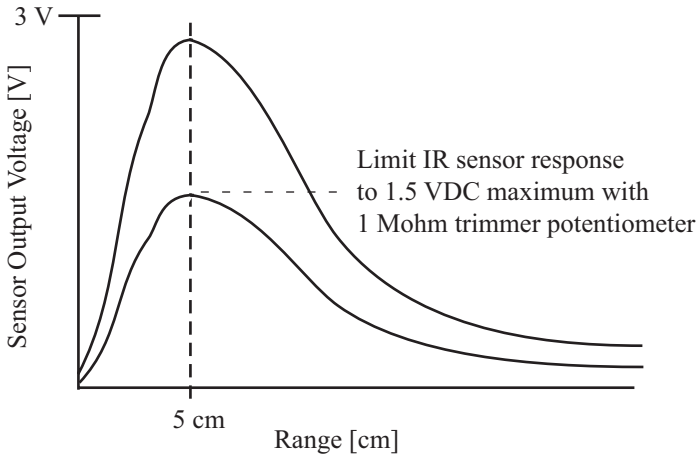


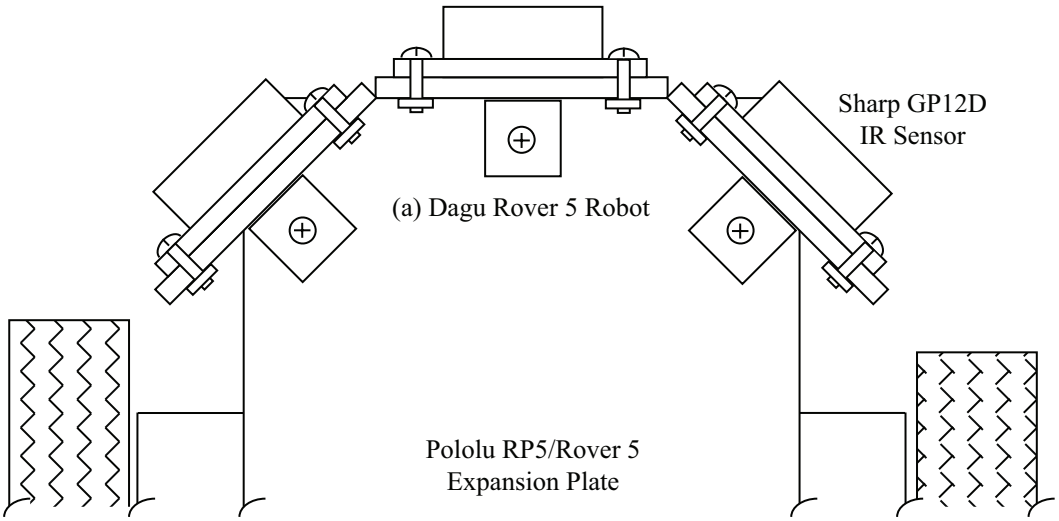
Figure 7.6: Sharp GP2Y0A21YKOF IR sensor profile.

7.5.3 CIRCUIT DIAGRAM-ARDUINO UNO

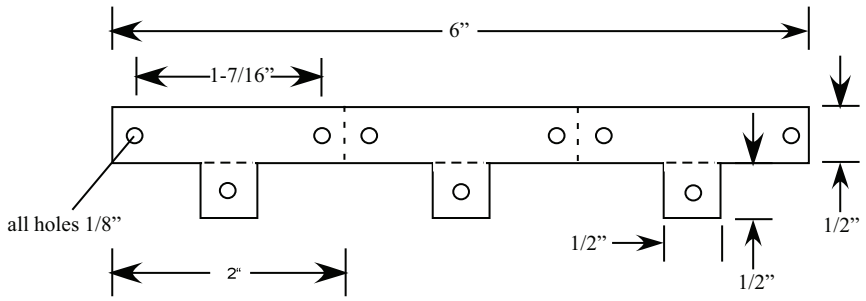
The circuit diagram for the robot is provided in Figure 7.9. The three IR sensors (left, middle, and right) are mounted on the leading edge of the robot to detect maze walls. The output from the sensors is fed to three Arduino UNO R3 ADC channels (ANALOG IN 0-2). The robot motors are driven by PWM channels (PWM: DIGITAL 11 and PWM: DIGITAL 10). The Arduino UNO R3 is interfaced to the motors via a Darlington NPN transistor (TIP120) with enough drive capability to handle the maximum current requirements of the motor. Since the motor power supply is at 9 VDC and the motors are rated at 7.2 VDC, three 1N4001 diodes are placed in series with the motor. This reduces the supply voltage to the motor to be approximately 6.9 VDC. The robot is powered by a 9 VDC power supply which is fed to a 5 VDC voltage regulator. To save on battery expense, it is recommended to use a 9 VDC, 2A rated inexpensive, wall-mount power supply. A power umbilical of braided wire may be used to provide power to the robot while navigating about the maze.

7.5.4 CIRCUIT DIAGRAM – ATMEGA328

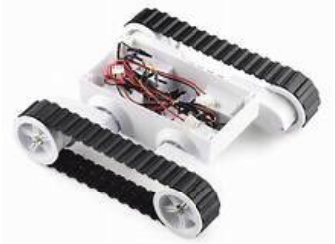
The circuit diagram for the robot is provided in Figure 7.10. The three IR sensors (left, middle, and right) are mounted on the leading edge of the robot to detect maze walls. The output from the sensor is fed to three ADC channels (PORTC[2:0]). The robot motors will be driven by PWM channels A and B (OC1A and OC1B). The microcontroller is interfaced to the motors via a Darlington transistor with enough drive capability to handle the maximum current requirements of the motor. Since the motor power supply is at 9 VDC and the motors are rated at 7.2 VDC, three 1N4001 diodes are placed in series with the motor. The robot is powered by a 9 VDC battery which is fed to a 5 VDC voltage regulator. Alternatively, a 9 VDC power supply



(a) Dagu Rover 5 Robot



(b) Construction Details for Sensor Bracket



(c) Dagu Rover 5-1 Tracked Robot

Figure 7.7: Dagu Rover 5 robot platform modified with three IR sensors.

	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 7.8: Truth table for robot action.

rated at several amps may be used in place of the 9 VDC battery. The supply may be connected to the robot via a flexible umbilical cable.

7.5.5 STRUCTURE CHART

The structure chart for the robot project is provided in Figure 7.11.

7.5.6 UML ACTIVITY DIAGRAMS

The UML activity diagram for the robot is provided in Figure 7.12.

7.5.7 MICROCONTROLLER CODE – ARDUINO UNO

The control algorithm begins with Arduino UNO R3 pin definitions. Variables are then declared for the readings from the three IR sensors. The two required Arduino functions follow: setup() and loop(). In the setup() function, Arduino UNO R3 pins are declared as output. The loop() begins by reading the current value of the three IR sensors. The 512 threshold value corresponds to a desired IR sensor range. This value may be adjusted to change the range at which the maze wall is detected. The IR sensor readings are followed by an eight part if-else if statement. The statement contains a part for each row of the truth table provided in Figure 7.8. For a given configuration of sensed walls, the appropriate wall detection LEDs are illuminated followed by commands to activate the motors (analogWrite) and illuminate the appropriate turn signals. The analogWrite command issues a signal from 0–5 VDC by sending a constant from 0–255 using pulse width modulation (PWM) techniques. The turn signal commands provide two actions: the appropriate turns signals are flashed and a 1.5 s total delay is provided. This provides the robot 1.5 s to render a turn. This delay may need to be adjusted during the testing phase.

```
//*********************************************************************
                                    //analog input pins
#define left_IR_sensor   A0         //analog pin - left IR sensor
```

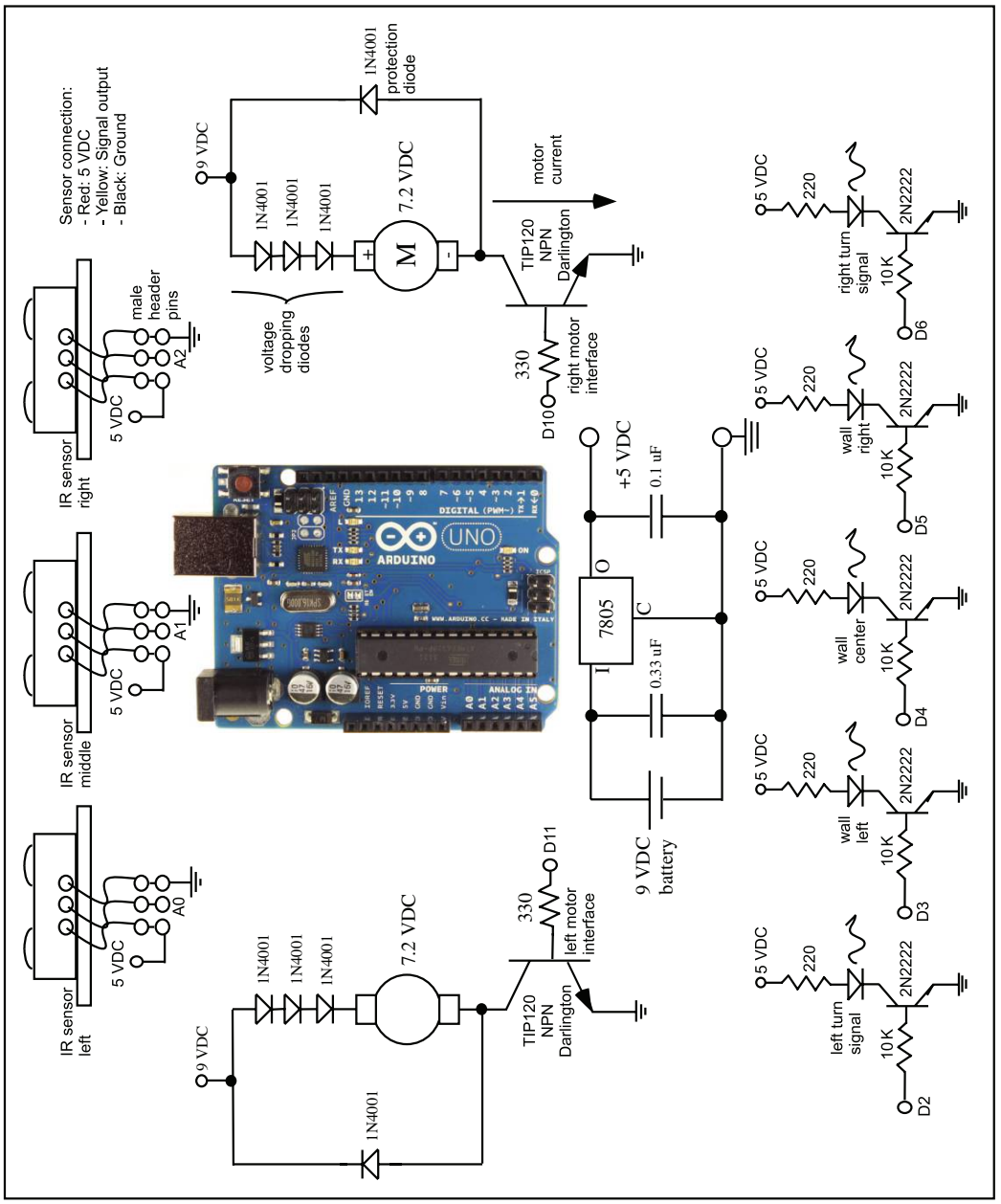


Figure 7.9: Robot circuit diagram. (UNO R3 illustration used with permission of the Arduino Team (CC BY-NC-SA) www.arduino.cc.)

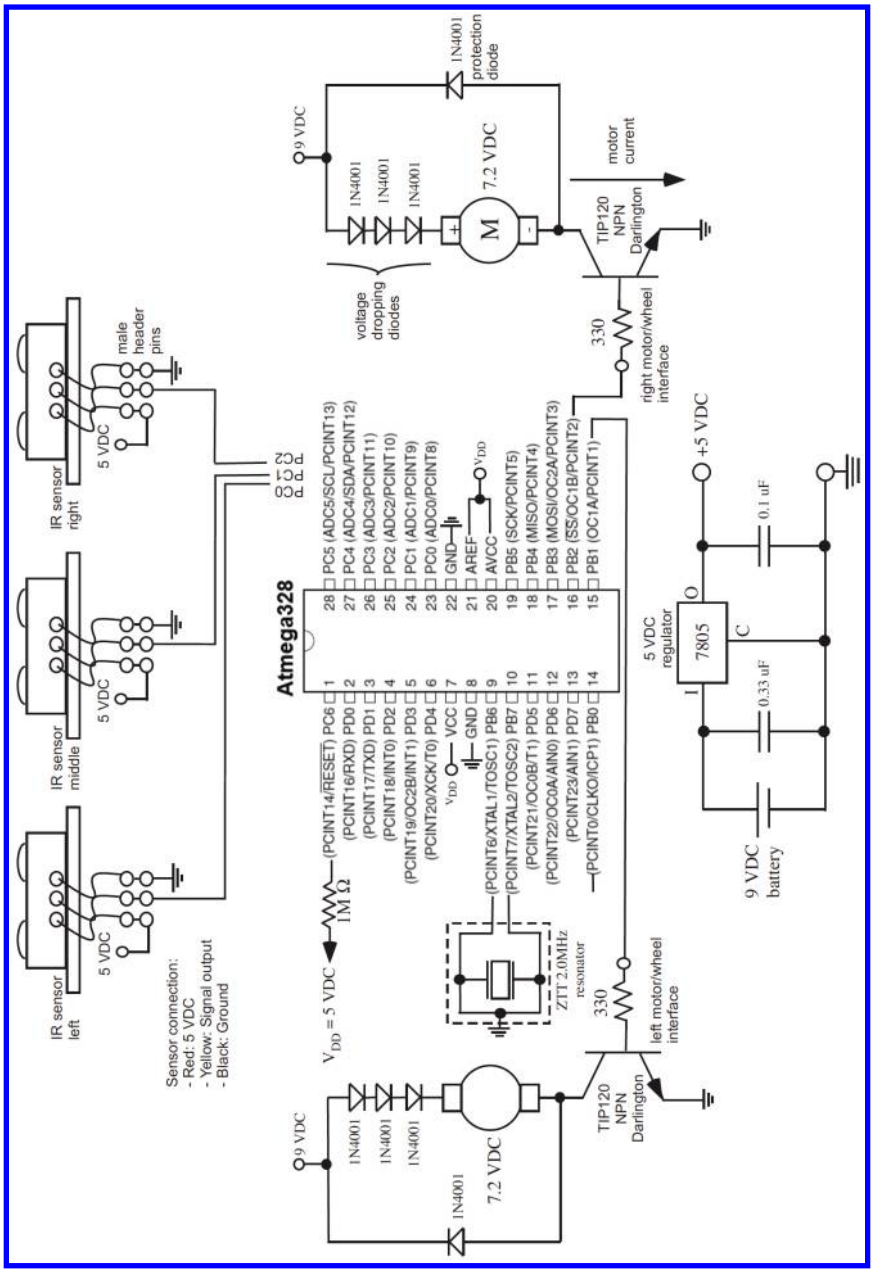


Figure 7.10: Robot circuit diagram.

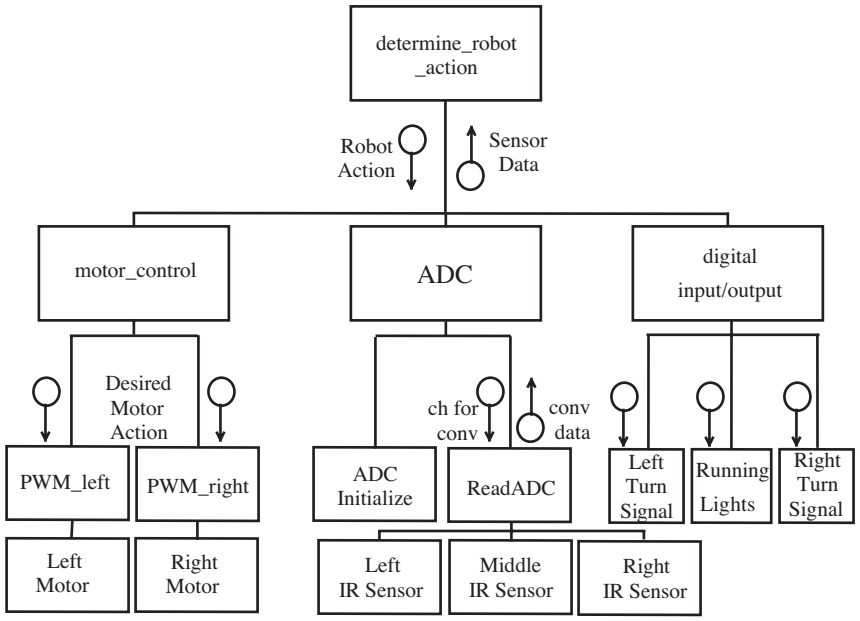


Figure 7.11: Robot structure diagram.

```
#define center_IR_sensor A1          //analog pin - center IR sensor
#define right_IR_sensor  A2          //analog pin - right IR sensor

                                     //digital output pins
                                     //LED indicators - wall detectors
#define wall_left         3          //digital pin - wall_left
#define wall_center       4          //digital pin - wall_center
#define wall_right        5          //digital pin - wall_right

                                     //LED indicators - turn signals
#define left_turn_signal  2          //digital pin - left_turn_signal
#define right_turn_signal 6          //digital pin - right_turn_signal

                                     //motor outputs
#define left_motor       11          //digital pin - left_motor
#define right_motor      10          //digital pin - right_motor
```

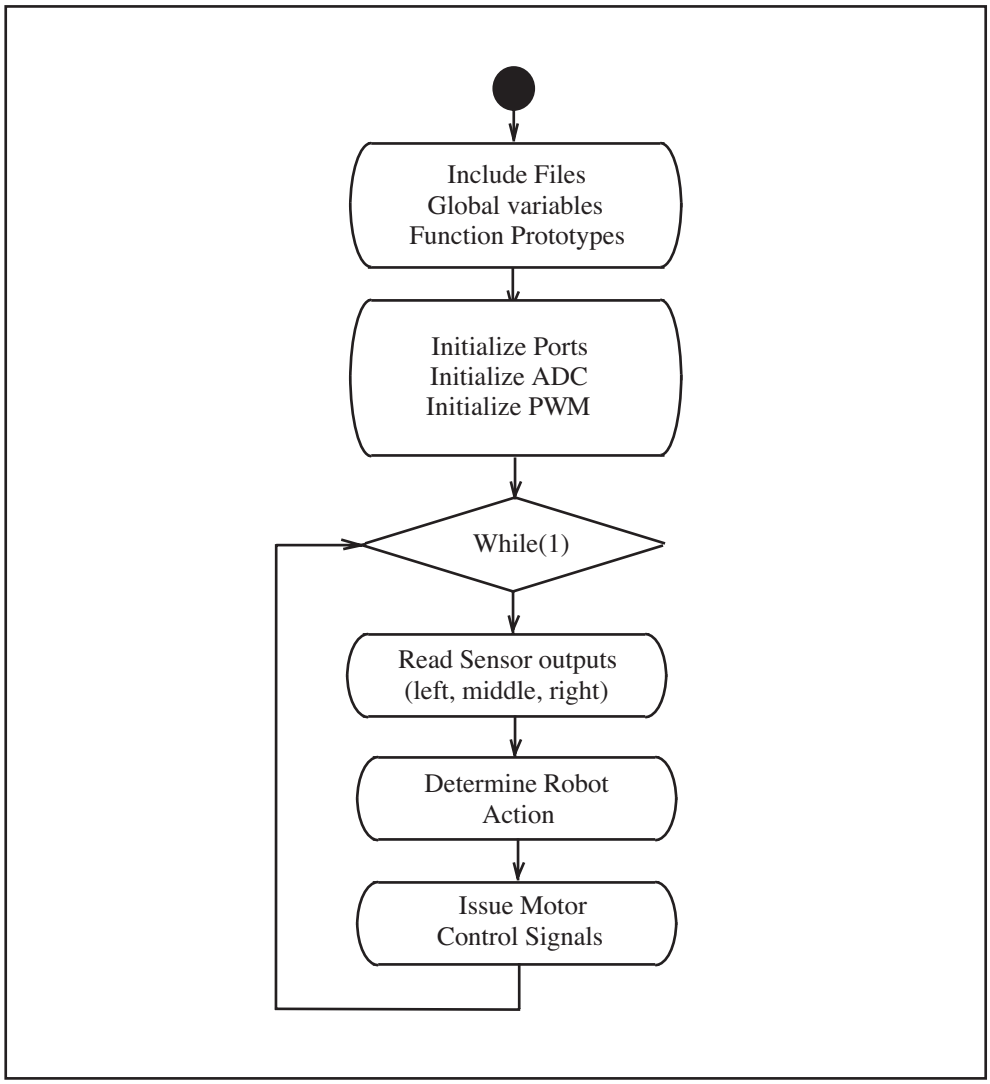


Figure 7.12: Robot UML activity diagram.

```
int left_IR_sensor_value;            //variable for left IR sensor
int center_IR_sensor_value;          //variable for center IR sensor
int right_IR_sensor_value;           //variable for right IR sensor

void setup()
  {
                                     //LED indicators - wall detectors
  pinMode(wall_left,   OUTPUT);      //configure pin 1 for digital output
  pinMode(wall_center, OUTPUT);      //configure pin 2 for digital output
  pinMode(wall_right,  OUTPUT);      //configure pin 3 for digital output

                                     //LED indicators - turn signals
  pinMode(left_turn_signal,OUTPUT);  //configure pin 0 for digital output
  pinMode(right_turn_signal,OUTPUT); //configure pin 4 for digital output

                                     //motor outputs - PWM
  pinMode(left_motor,  OUTPUT);      //config pin 11 for digital output
  pinMode(right_motor, OUTPUT);      //config pin 10 for digital output
  }

void loop()
  {
                                     //read analog output from IR sensors
   left_IR_sensor_value   = analogRead(left_IR_sensor);
   center_IR_sensor_value = analogRead(center_IR_sensor);
   right_IR_sensor_value  = analogRead(right_IR_sensor);

   //robot action table row 0
   if((left_IR_sensor_value < 512)&&(center_IR_sensor_value < 512)&&
      (right_IR_sensor_value < 512))
     {
                                           //wall detection LEDs
     digitalWrite(wall_left,   LOW);       //turn LED off
     digitalWrite(wall_center, LOW);       //turn LED off
     digitalWrite(wall_right,  LOW);       //turn LED off
                                           //motor control
     analogWrite(left_motor,  128);        //0 (off) to
                                           //255 (full speed)
     analogWrite(right_motor, 128);        //0 (off) to
```

```
                                        //255 (full speed)
                                        //turn signals
  digitalWrite(left_turn_signal,  LOW);  //turn LED off
  digitalWrite(right_turn_signal, LOW);  //turn LED off
  delay(500);                           //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);  //turn LED off
  digitalWrite(right_turn_signal, LOW);  //turn LED off
  delay(500);                           //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);  //turn LED off
  digitalWrite(right_turn_signal, LOW);  //turn LED off
  delay(500);                           //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);  //turn LED off
  digitalWrite(right_turn_signal, LOW);  //turn LED off
  analogWrite(left_motor, 0);           //turn motor off
  analogWrite(right_motor,0);           //turn motor off
  }

//robot action table row 1
else if((left_IR_sensor_value < 512)&&(center_IR_sensor_value < 512)&&
        (right_IR_sensor_value > 512))
  {
                                        //wall detection LEDs
  digitalWrite(wall_left,   LOW);       //turn LED off
  digitalWrite(wall_center, LOW);       //turn LED off
  digitalWrite(wall_right,  HIGH);      //turn LED on
                                        //motor control
  analogWrite(left_motor,  128);        //0 (off) to
                                        //255 (full speed)
  analogWrite(right_motor, 128);        //0 (off) to
                                        //255 (full speed)
                                        //turn signals
  digitalWrite(left_turn_signal,  LOW);  //turn LED off
  digitalWrite(right_turn_signal, LOW);  //turn LED off
  delay(500);                           //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);  //turn LED off
  digitalWrite(right_turn_signal, LOW);  //turn LED off
  delay(500);                           //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);  //turn LED off
  digitalWrite(right_turn_signal, LOW);  //turn LED off
```

```
  delay(500);                              //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);    //turn LED off
  digitalWrite(right_turn_signal, LOW);    //turn LED off
  analogWrite(left_motor, 0);              //turn motor off
  analogWrite(right_motor,0);              //turn motor off
  }

//robot action table row 2
else if((left_IR_sensor_value < 512)&&(center_IR_sensor_value > 512)&&
        (right_IR_sensor_value < 512))
  {
                                           //wall detection LEDs
  digitalWrite(wall_left,   LOW);          //turn LED off
  digitalWrite(wall_center, HIGH);         //turn LED on
  digitalWrite(wall_right,  LOW);          //turn LED off
                                           //motor control
  analogWrite(left_motor,  128);           //0 (off) to
                                           //255 (full speed)
  analogWrite(right_motor, 0);             //0 (off) to
                                           //255 (full speed)
                                           //turn signals
  digitalWrite(left_turn_signal,  LOW);    //turn LED off
  digitalWrite(right_turn_signal, HIGH);   //turn LED on
  delay(500);                              //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);    //turn LED off
  digitalWrite(right_turn_signal, LOW);    //turn LED off
  delay(500);                              //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);    //turn LED off
  digitalWrite(right_turn_signal, HIGH);   //turn LED on
  delay(500);                              //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);    //turn LED off
  digitalWrite(right_turn_signal, LOW);    //turn LED off
  analogWrite(left_motor, 0);              //turn motor off
  analogWrite(right_motor,0);              //turn motor off
  }

//robot action table row 3
else if((left_IR_sensor_value < 512)&&(center_IR_sensor_value > 512)&&
        (right_IR_sensor_value > 512))
```

```
  {
                                        //wall detection LEDs
  digitalWrite(wall_left,   LOW);       //turn LED off
  digitalWrite(wall_center, HIGH);      //turn LED on
  digitalWrite(wall_right,  HIGH);      //turn LED on
                                        //motor control
  analogWrite(left_motor,  0);          //0 (off) to
                                        //255 (full speed)
  analogWrite(right_motor, 128);        //0 (off) to
                                        //255 (full speed)
                                        //turn signals
  digitalWrite(left_turn_signal,  HIGH); //turn LED on
  digitalWrite(right_turn_signal, LOW);  //turn LED off
  delay(500);                           //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);  //turn LED off
  digitalWrite(right_turn_signal, LOW);  //turn LED off
  delay(500);                           //delay 500 ms
  digitalWrite(left_turn_signal,  HIGH); //turn LED on
  digitalWrite(right_turn_signal, LOW);  //turn LED off
  delay(500);                           //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);  //turn LED off
  digitalWrite(right_turn_signal, LOW);  //turn LED off
  analogWrite(left_motor, 0);           //turn motor off
  analogWrite(right_motor,0);           //turn motor off
  }

//robot action table row 4
else if((left_IR_sensor_value > 512)&&(center_IR_sensor_value < 512)&&
        (right_IR_sensor_value < 512))
  {
                                        //wall detection LEDs
  digitalWrite(wall_left,   HIGH);      //turn LED on
  digitalWrite(wall_center, LOW);       //turn LED off
  digitalWrite(wall_right,  LOW);       //turn LED off
                                        //motor control
  analogWrite(left_motor,  128);        //0 (off) to
                                        //255 (full speed)
  analogWrite(right_motor, 128);        //0 (off) to
                                        //255 (full speed)
```

```
                                          //turn signals
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  analogWrite(left_motor, 0);             //turn motor off
  analogWrite(right_motor,0);             //turn motor off
  }

//robot action table row 5
else if((left_IR_sensor_value > 512)&&(center_IR_sensor_value < 512)&&
        (right_IR_sensor_value > 512))
  {
                                          //wall detection LEDs
  digitalWrite(wall_left,   HIGH);        //turn LED on
  digitalWrite(wall_center, LOW);         //turn LED off
  digitalWrite(wall_right,  HIGH);        //turn LED on
                                          //motor control
  analogWrite(left_motor,  128);          //0 (off) to
                                          //255 (full speed)
  analogWrite(right_motor, 128);          //0 (off) to
                                          //255 (full speed)
                                          //turn signals
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
```

```
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  analogWrite(left_motor, 0);             //turn motor off
  analogWrite(right_motor,0);             //turn motor off
  }

//robot action table row 6
else if((left_IR_sensor_value > 512)&&(center_IR_sensor_value > 512)&&
        (right_IR_sensor_value < 512))
  {
                                          //wall detection LEDs
  digitalWrite(wall_left,   HIGH);        //turn LED on
  digitalWrite(wall_center, HIGH);        //turn LED on
  digitalWrite(wall_right,  LOW);         //turn LED off
                                          //motor control
  analogWrite(left_motor,  128);          //0 (off) to
                                          //255 (full speed)
  analogWrite(right_motor, 0);            //0 (off) to
                                          //255 (full speed)
                                          //turn signals
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, HIGH);  //turn LED on
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, LOW);   //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED off
  digitalWrite(right_turn_signal, HIGH);  //turn LED off
  delay(500);                             //delay 500 ms
  digitalWrite(left_turn_signal,  LOW);   //turn LED OFF
  digitalWrite(right_turn_signal, LOW);   //turn LED OFF
  analogWrite(left_motor, 0);             //turn motor off
  analogWrite(right_motor,0);             //turn motor off
  }

//robot action table row 7
else if((left_IR_sensor_value > 512)&&(center_IR_sensor_value > 512)&&
        (right_IR_sensor_value > 512))
  {
```

```
                                          //wall detection LEDs
    digitalWrite(wall_left,   HIGH);      //turn LED on
    digitalWrite(wall_center, HIGH);      //turn LED on
    digitalWrite(wall_right,  HIGH);      //turn LED on
                                          //motor control
    analogWrite(left_motor,  128);        //0 (off) to
                                          //255 (full speed)
    analogWrite(right_motor, 0);          //0 (off) to
                                          //255 (full speed)
                                          //turn signals
    digitalWrite(left_turn_signal,  LOW);  //turn LED off
    digitalWrite(right_turn_signal, HIGH); //turn LED on
    delay(500);                           //delay 500 ms
    digitalWrite(left_turn_signal,  LOW);  //turn LED off
    digitalWrite(right_turn_signal, LOW);  //turn LED off
    delay(500);                           //delay 500 ms
    digitalWrite(left_turn_signal,  LOW);  //turn LED off
    digitalWrite(right_turn_signal, HIGH); //turn LED on
    delay(500);                           //delay 500 ms
    digitalWrite(left_turn_signal,  LOW);  //turn LED off
    digitalWrite(right_turn_signal, LOW);  //turn LED off
    analogWrite(left_motor, 0);           //turn motor off
    analogWrite(right_motor,0);           //turn motor off
    }
}

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

7.5.8 MICROCONTROLLER CODE – ATMEGA328

Provided below is the basic framework for the code. As illustrated in the Robot UML activity diagram, the control algorithm initializes various ATmega328 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 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.

```
//*************************************************************
//robot_control
//Note: ATmega328 is clocked by an external 2.0 MHz resonator
//*************************************************************

#include<iom328v.h>          //ATmega328 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_32_7ms_interrupts);
void init_timer2_ovf_interrupt(void);
void timer2_interrupt_isr(void);
void initialize_ports(void);
void determine_robot_action(void);

//interrupt handler definition
#pragma interrupt_handler timer2_interrupt_isr:10

//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)
//Note: 1: output, 0: input
//*************************************************************

void initialize_ports(void)
{
DDRB = 0xFF;
DDRC = 0xF8;                  //PORTC[2:0] input
DDRD = 0xFF;
}

//*************************************************************
//void determine_robot_action(void)
//Note: 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)&&
   (right_IR_voltage < 2.5))
  {
  PWM_right();
  }

  :                           //insert other robot action cases
  :
  :
}

//*************************************************************
//void ADC_values(void)
```

```
//  PORTC[0] - Left IR Sensor
//  PORTC[1] - Center IR Sensor
//  PORTC[2] - Right IR Sensor
//*************************************************************

void ADC_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 = 2 MHz/510
                              //freq = 3.922 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(31);                    //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 = 2 MHz/510
                                //freq = 3.922 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(31);                     //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
```

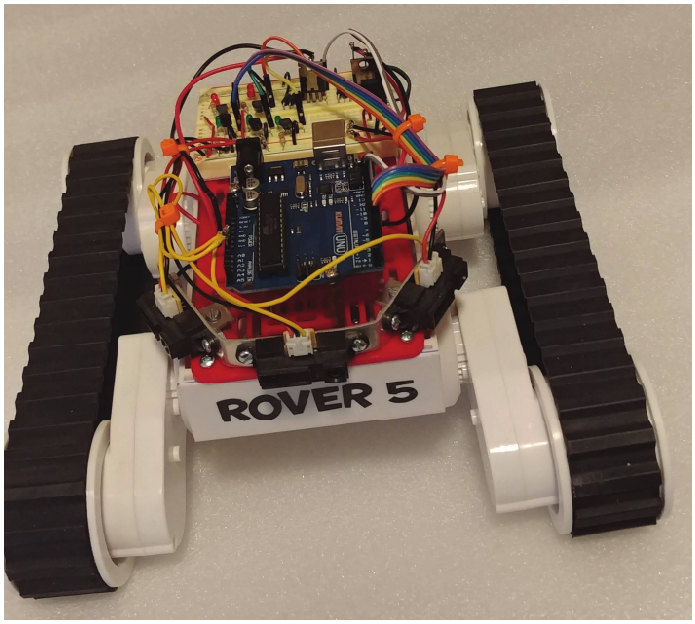
```
//    motor.
//  - The length of the delay controls the amount of time the
//    motors are powered.
//*****************************************************************

void PWM_right(void)
{
TCCR1A = 0xA1;                    //freq = resonator/510 = 2 MHz/510
                                  //freq = 3.922 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(31);                       //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 int binary_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_32_7ms_interrupts)
{
TCNT2 = 0x00;                   //reset timer2
input_delay = 0;                //reset timer2 overflow counter
while(input_delay <= number_of_32_7ms_interrupts)
  {
  ;                             //wait number of interrupts
```

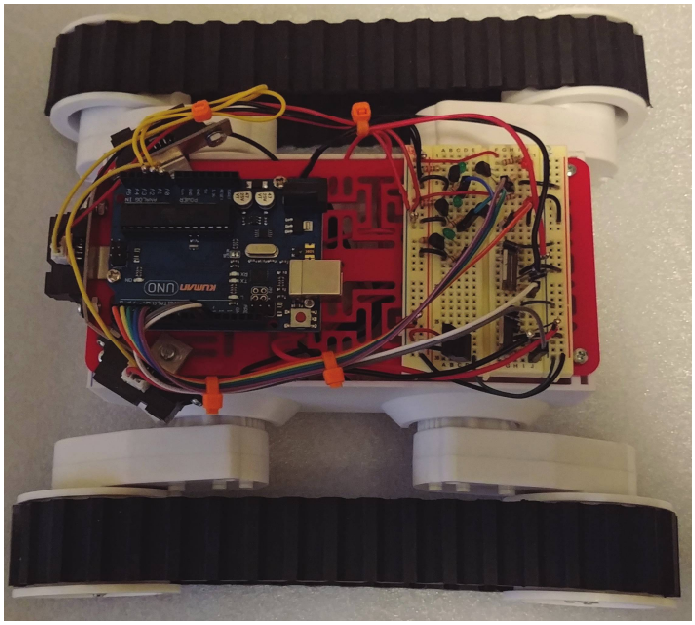
```
  }
}

//************************************************************
// void init_timer2_ovf_interrupt(void)
//
// - ATmega328 clock source: 2 MHz
// - Divided by 256: 7,812 KHz clock source to Timer 2
// - Timer 2 receives a clock tick every 128 microseconds
// - Overflows every 256 counts or 32.8 ms
//************************************************************

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 32.7 ms
TIMSK2 = 0x01;   //enable timer2 overflow interrupt
asm("SEI");      //enable global interrupt
}

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

void timer2_interrupt_isr(void)
{
input_delay++;                    //increment overflow counter
}

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

Provided in Figured 7.13 are images of the assembled robot using the Arduino UNO R3.

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 ATmega328 as specified in Figure 7.10. 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 ATmega328 may be mounted to the robot platform and connected to the motors. Full-up testing in the maze may commence. The design provided is very basic. The chapter homework assignments provide opportunity to extend the design with additional features.



(a) Dagu Rover 5-1 Front View



(b) Dagu Rover 5-1 Top View

Figure 7.13: Dagu Rover 5 robot platform modified with three IR sensors.

7.6 SUMMARY

In this chapter, we discussed the design process, related tools, and applied the process to a real-world design. It is essential to follow a systematic, disciplined approach to embedded systems design to successfully develop a prototype that meets established requirements.

7.7 REFERENCES

[1] M. Anderson. Help wanted: Embedded engineers—Why the United States is losing its edge in embedded systems, *IEEE–USA Today's Engineer*, February 2008. 223

[2] S. F. Barrett and D. J. Pack. *Microcontrollers Fundamentals for Engineers and Scientists*, Morgan & Claypool Publishers, 2006. DOI: 10.2200/s00025ed1v01y200605dcs001. 226

[3] S. F. Barrett and D. J. Pack. *Atmel® AVR Microcontroller Primer Programming and Interfacing*, Morgan & Claypool Publishers, 2008. DOI: 10.2200/s00100ed1v01y200712dcs015.

[4] S. F. Barrett. *Embedded Systems Design with the Atmel® AVR Microcontroller*, Morgan & Claypool Publishers, 2010. DOI: 10.2200/s00138ed1v01y200910dcs024.

[5] M. Fowler and K. Scott, *UML Distilled—A Brief Guide to the Standard Object Modeling Language*, 2nd ed., Boston, Addison-Wesley, 2000. 227

[6] N. Dale and S. C. Lilly. *Pascal Plus Data Structures*, 4th ed., Jones and Bartlett, Englewood Cliffs, NJ, 1995. 228

[7] *Microchip ATmega328 PB AVR Microcontroller with Core Independent Peripherals and Pico Power Technology DS40001906C*, Microchip Technology Incorporation, 2018. www.microchip.com

[8] S. F. Barrett and D. J. Pack. *Microchip AVR Microcontroller Primer: Programming and Interfacing*, Morgan & Claypool Publishers, 2019.

[9] *Microchip ATmega328 PB AVR Microcontroller with Core Independent Peripherals and Pico Power Technology DS40001906C*, Microchip Technology Incorporation, 2018. www.microchip.com

[10] S. F. Barrett and D. J. Pack. *Microchip AVR Microcontroller Primer: Programming and Interfacing*, 3rd ed., Morgan & Claypool Publishers, 2019.

7.8 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. Provide a UML activity diagram and a structure chart for Automated Fan Cooling System.
10. What is the purpose of the 4N35 optical coupler in the motor control circuit?
11. Modify the Rover 5 design to include PWM turning commands such that the PWM duty cycle and the length of the motors are on are sent in as variables to the function.
12. Modify the Rover 5 design to include 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.
13. Modify the Rover 5 design to include a function for reversing the robot.
14. Investigate the use of a Rover 5-2 tracked robot (two motors, two motor encoders) to count the number of wheel rotations.
15. Write an interrupt driven function to count wheel rotations for the Rover 5-2 tracked robot.

Author's Biography

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 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 Vice Provost of Undergraduate Education at the University of Wyoming and Professor of Electrical and Computer Engineering. 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) Professional Engineers in Higher Education, Engineering Education Excellence Award.

Index