Arduino Microcontroller Processing for Everyone! Part II

Synthesis Lectures on Digital Circuits and Systems

Arduino Microcontroller: Processing for Everyone! Part II

Steven F. Barrett

www.morganclaypool.com

ISBN: 9781608454372 paperback
ISBN: 9781608454884 ebook

DOI 10.2200/S00283ED1V01Y201005DCS029

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

Lecture #29
Series Editor: Mitchell A. Thornton, *Southern Methodist University*
Series ISSN
Synthesis Lectures on Digital Circuits and Systems
Print 1932-3166 Electronic 1932-3174

Arduino Microcontroller Processing for Everyone! Part II

Steven F. Barrett
University of Wyoming, Laramie, WY

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS #29

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. This book is intended 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 meet this wide audience, the book has been divided into sections to satisfy the need of each reader. The book contains many software and hardware examples to assist the reader in developing a wide variety of systems. For the examples, the Arduino Duemilanove and the Atmel ATmega328 is employed as the target processor.

KEYWORDS

Arduino microcontroller, Arduino Duemilanove, Atmel microcontroller, Atmel AVR, ATmega328, microcontroller interfacing, embedded systems design

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. There approach was to openly share details of microcontroller-based hardware design platforms to stimulate the sharing of ideas and innovation. This concept has been popular in the software world for many years.

This book is written for a number of audiences. First, in keeping with the Arduino concept, the book is written for practitioners of the arts (design students, artists, photographers, etc.) who may need processing power in a project but do not have an in depth engineering background. Second, the book is written for middle school and senior high school students who may need processing power for a school or science fair project. Third, we write for engineering students who require processing power for their senior design project but do not have the background in microcontroller-based applications commonly taught in electrical and computer engineering curricula. Finally, the book provides practicing scientists and engineers an advanced treatment of the Atmel AVR microcontroller.

APPROACH OF THE BOOK

To encompass such a wide range of readers, we have divided the book into several portions to address the different readership. Chapters 1 through 2 are intended for novice microcontroller users. Chapter 1 provides a review of the Arduino concept, a description of the Arduino Duemilanove development board, and a brief review of the features of the Duemilanove's host processor, the Atmel ATmega 328 microcontroller. Chapter 2 provides an introduction to programming for the novice programmer. Chapter 2 also introduces the Arduino Development Environment and how to program sketches. It also serves as a good review for the seasoned developer.

Chapter 3 provides an introduction to embedded system design processes. It provides a systematic, step-by-step approach on how to design complex systems in a stress free manner.

Chapters 4 through 8 provide detailed engineering information on the ATmega328 microcontroller and advanced interfacing techniques. These chapters are intended for engineering students and practicing engineers. However, novice microcontroller users will find the information readable and well supported with numerous examples.

The final chapter provides a variety of example applications for a wide variety of skill levels.

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 book. I would also like to thank Joel Claypool of Morgan & Claypool Publishers who has supported a number of writing projects of Daniel Pack and I over the last several years. He also provided permission to include portions of background information on the Atmel line of AVR microcontrollers in this book from several of our previous projects. I would also like to thank Sparkfun Electronics of Boulder, Colorado; Atmel Incorporated; the Arduino team; and ImageCraft of Palo Alto, California for use of pictures and figures used within the book.

I would like to dedicate this book to my close friend and writing partner Dr. Daniel Pack, Ph.D., P.E. Daniel elected to "sit this one out" because of a thriving research program in unmanned aerial vehicles (UAVs). Much of the writing is his from earlier Morgan & Claypool projects. 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 six more since then). Daniel is a good father, good son, good husband, brilliant engineer, 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.

Finally, I would like to thank my wife and best friend of many years, Cindy.

Laramie, Wyoming, May 2010

Steve Barrett

CHAPTER 5

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 Duemilanove 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.
- Describe the operation of a digital-to-analog converter (DAC).

5.1 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. In the first section, we discuss the conversion process itself, followed by a presentation of the successive-approximation 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 Arduino Development Environment and C. We conclude the chapter with a discussion of the DAC process and interface a multi-channel DAC to the ATmega328. We also discuss the Arduino Development Environment built-in features that allow generation of an output analog signal via pulse width modulation (PWM) techniques. Throughout the chapter, we provide detailed examples.

5.2 SAMPLING, QUANTIZATION AND ENCODING

In this subsection, 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 [Barrett and Pack]. 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 'snap shots' 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 signals. 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 'snap shots' 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 analog-to-digital conversion, 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 5.1 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 to 5 volts. In Figure 5.1, 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 5.1. 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 5.1.

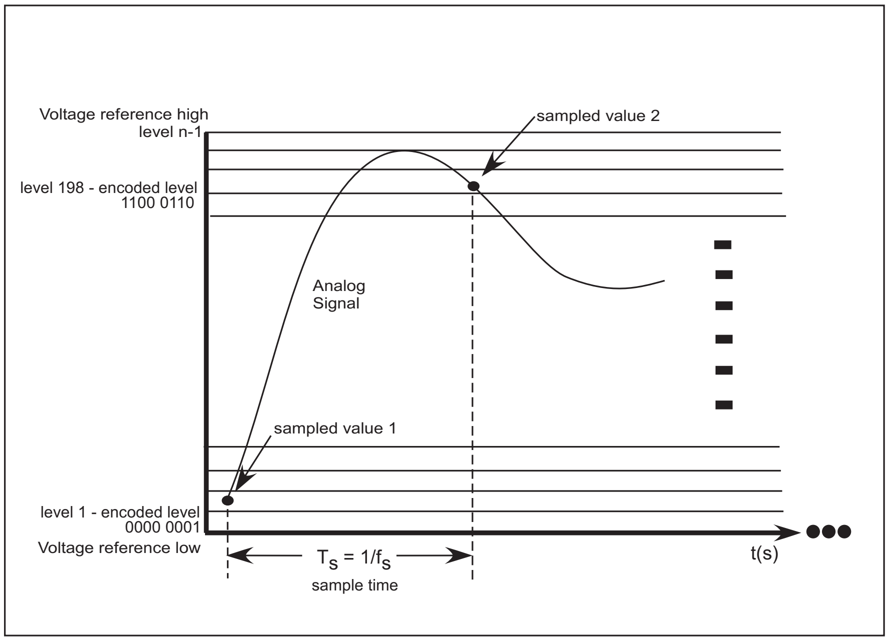


Figure 5.1: Sampling, quantization, and encoding.

5.2.1 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:

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

for the ATmega328, the resolution is:

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

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 8KHz sampling rate (f_s) and you are using 10 bits for the quantization levels for each sampled signal.[1] This means the voice conversation will be sampled every 125 microseconds (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 x 8000 x 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 versus the required system resolution.

Dynamic range. You will also encounter the term "dynamic range" when you consider finding appropriate analog-to-digital converters. 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 to 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.

5.3 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.

[1] 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.

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. Interface techniques are discussed in detail in Chapter 8.

5.3.1 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 to 5 VDC. Figure 5.2 shows the transducer interface circuit using an input transducer.

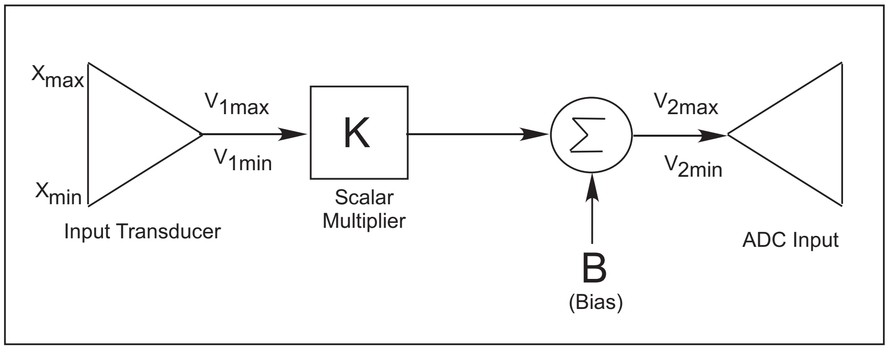


Figure 5.2: A block diagram of the signal conditioning for an analog-to-digital converter. The range of the sensor voltage output is mapped to the analog-to-digital converter input voltage range. The scalar multiplier maps the magnitudes of the two ranges and the bias voltage is used to align two limits.

The output of the input transducer is first scaled by constant K. In the figure, we use a microphone as the input transducer whose output ranges from -5 VDC to + 5 VDC. The input to the analog-to-digital converter ranges from 0 VDC to 5 VDC. The box with constant K maps the output range of the input transducer to the input range of the converter. Naturally, we need to multiply all input signals by 1/2 to accommodate the mapping. Once the range has been mapped, the signal now needs to be shifted. Note that the scale factor maps the output range of the input transducer as -2.5 VDC to +2.5 VDC instead of 0 VDC to 5 VDC. The second portion of the circuit 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:

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

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

The variable V_{1max} 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_{2max} to the input of the ADC converter [USAFA].

Similarly, The variable V_{1min} 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_{2min} to the input of the ADC converter.

Usually, the values of V_{1max} and V_{1min} are provided with the documentation for the transducer. Also, the values of V_{2max} and V_{2min} 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.

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

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

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

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

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

5.3.2 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.

5.3.2.1 The ideal operational amplifier

A generic ideal operational amplifier is illustrated in Figure 5.3. 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.

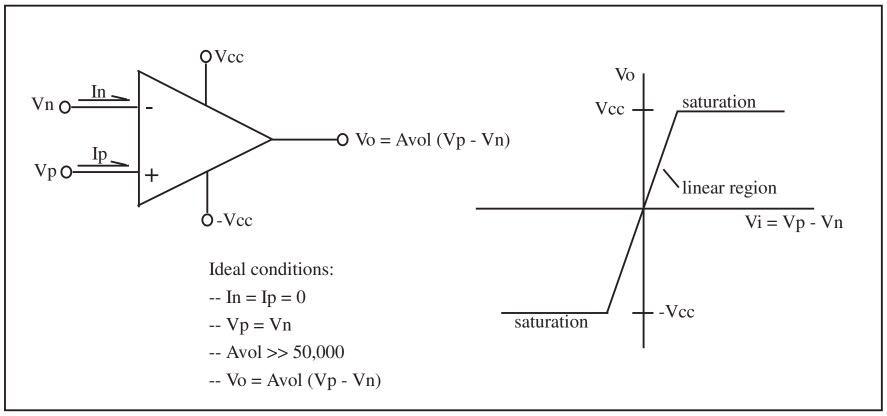


Figure 5.3: Ideal operational amplifier characteristics.

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 5.4 [Faulkenberry].

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:

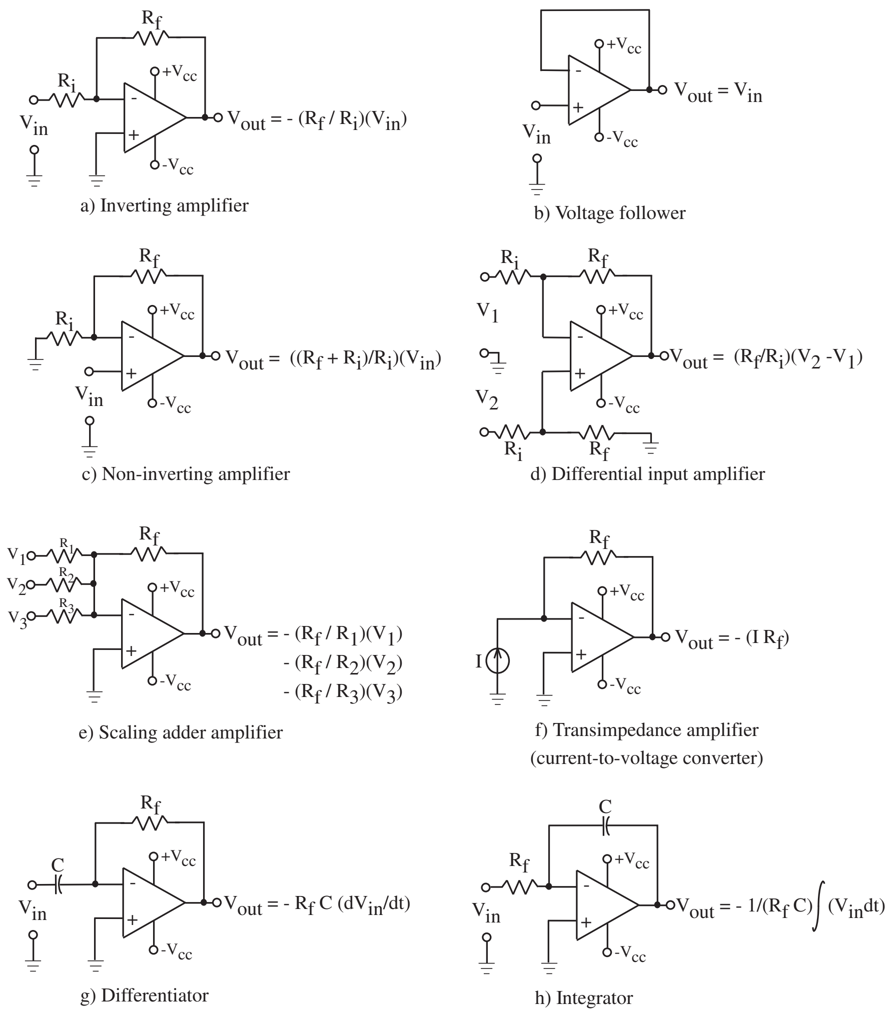

a) Inverting amplifier
$V_{out} = -(R_f / R_i)(V_{in})$
b) Voltage follower
$V_{out} = V_{in}$
c) Non-inverting amplifier
$V_{out} = ((R_f + R_i)/R_i)(V_{in})$
d) Differential input amplifier
$V_{out} = (R_f/R_i)(V_2 - V_1)$
e) Scaling adder amplifier
$V_{out} = -(R_f / R_1)(V_1) - (R_f / R_2)(V_2) - (R_f / R_3)(V_3)$
f) Transimpedance amplifier (current-to-voltage converter)
$V_{out} = -(I R_f)$
g) Differentiator
$V_{out} = -R_f C (dV_{in}/dt)$
h) Integrator
$V_{out} = -1/(R_f C)\int (V_{in} dt)$

Figure 5.4: Classic operational amplifier configurations. Adapted from [Faulkenberry].

- 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 5.5. This same analysis technique may be applied to all of the circuits in Figure 5.4 to arrive at the equations for Vout provided.

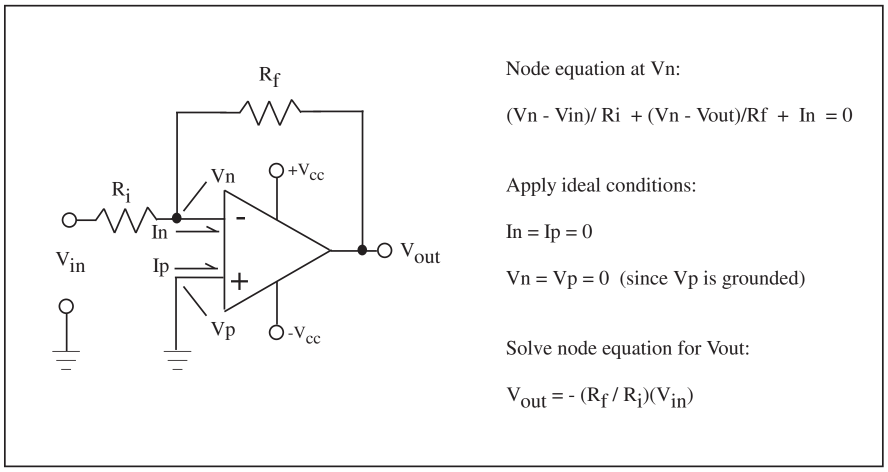

R_f
R_i
V_{in}
Vn
In
Ip
Vp
$+V_{cc}$
$-V_{cc}$
-
+
V_{out}
Node equation at Vn:
(Vn - Vin)/ Ri + (Vn - Vout)/Rf + In = 0
Apply ideal conditions:
In = Ip = 0
Vn = Vp = 0 (since Vp is grounded)
Solve node equation for Vout:
$V_{out} = -(R_f / R_i)(V_{in})$

Figure 5.5: Operational amplifier analysis for the non-inverting amplifier. Adapted from [Faulkenberry].

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 5.6 implements these values of K and B. The first stage provides an amplification of -2 due to the use of the non-inverting amplifier configuration. In the 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.

5.4 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 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.

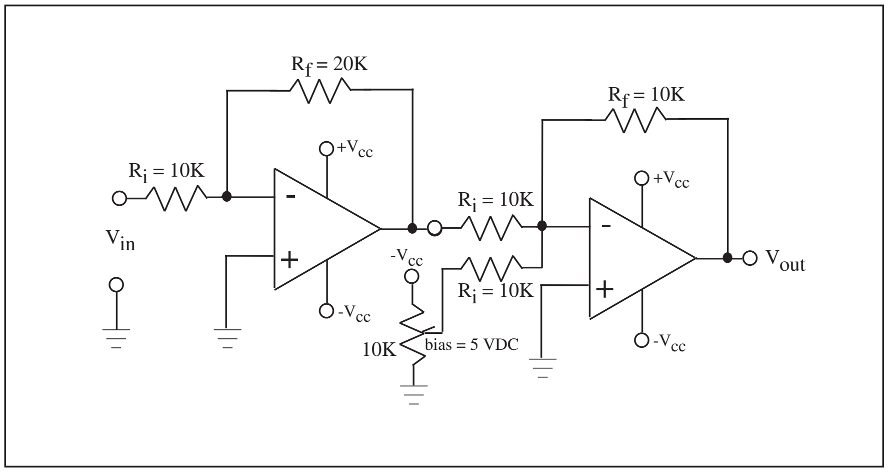


Figure 5.6: Operational amplifier implementation of the transducer interface design (TID) example circuit.

5.4.1 SUCCESSIVE-APPROXIMATION

The ATmega328 microcontroller is equipped with a successive-approximation ADC converter. The successive-approximation technique uses a digital-to-analog converter, 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 digital-to-analog converter, 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 5.7 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.

5.5 THE ATMEL ATMEGA328 ADC SYSTEM

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

- 10-bit resolution
- ±2 least significant bit (LSB) absolute accuracy
- 13 ADC clock cycle conversion time
- 6 multiplexed single ended input channels

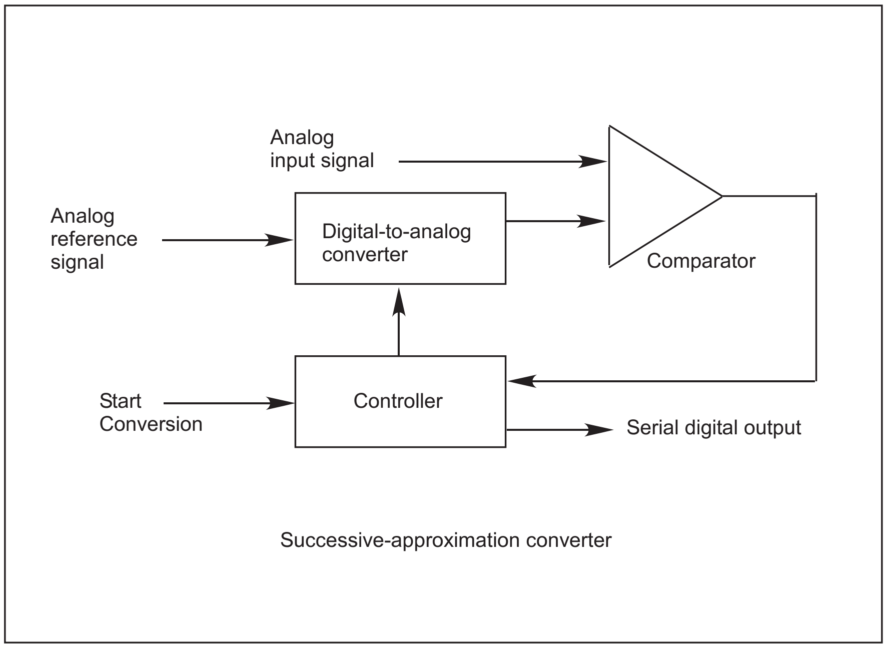


Figure 5.7: Successive-approximation ADC.

- Selectable right or left result justification
- 0 to Vcc ADC input voltage range

Let's discuss each feature in turn. The first feature of discussion is "10-bit resolution." Resolution is defined as:

$$Resolution = (V_{RH} - V_{RL})/2^b$$

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 ATmega328 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 [Atmel].

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 an six 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.

5.5.1 BLOCK DIAGRAM

The block diagram for the ATmega328 ADC conversion system is provided in Figure 5.8. The left edge of the diagram provides the external microcontroller pins to gain access to the ADC. The six analog input channels are provided at ADC[5:0] and the ADC reference voltage pins are provided at AREF and AVCC. The key features and registers of the ADC system previously discussed are included in the diagram.

5.5.2 REGISTERS

The key registers for the ADC system are shown in Figure 5.9. 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 [Atmel].

5.5.2.1 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 analog-to-digital converter 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

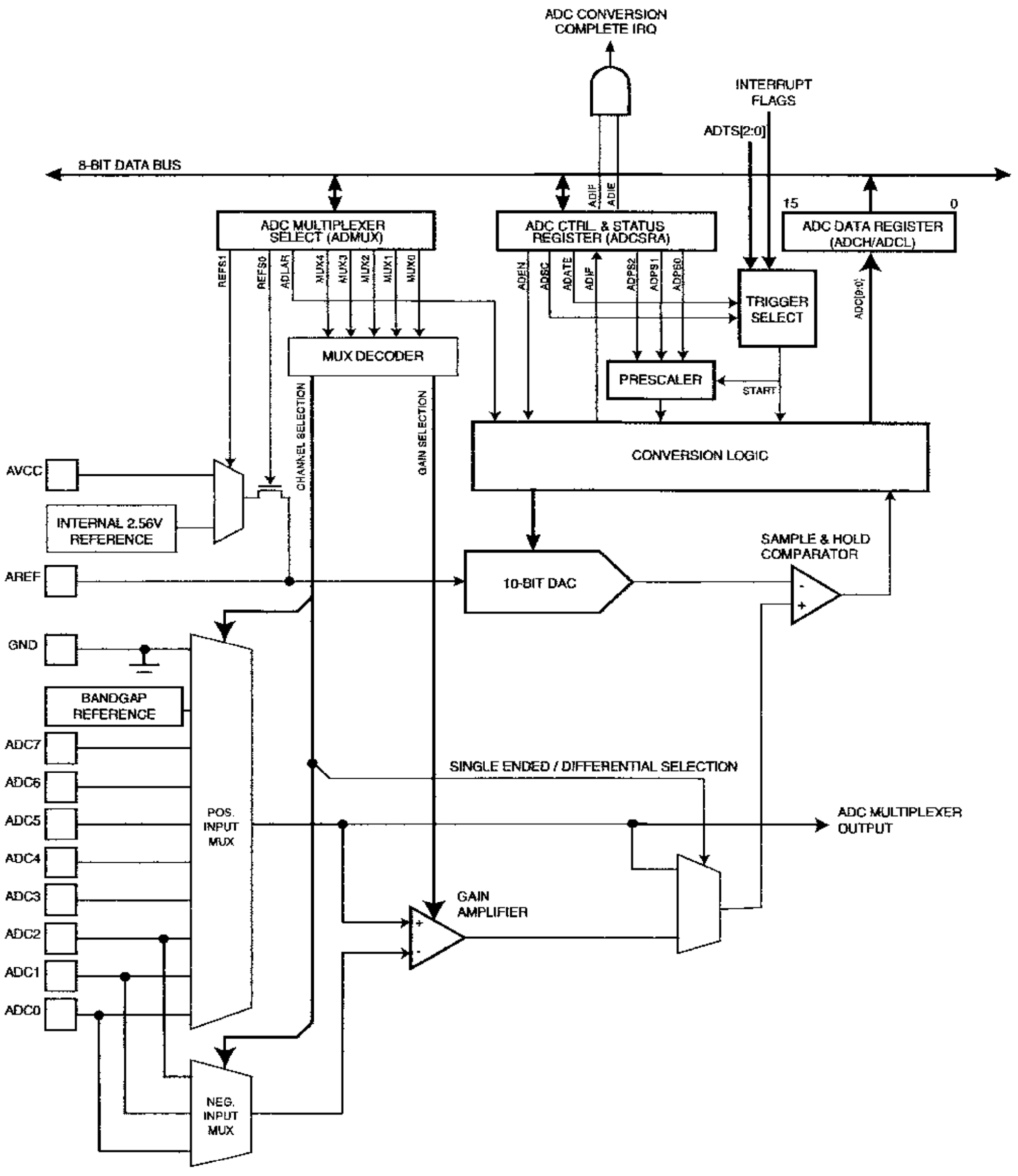


Figure 5.8: Atmel AVR ATmega328 ADC block diagram. (Figure used with permission of Atmel, Incorporated.)

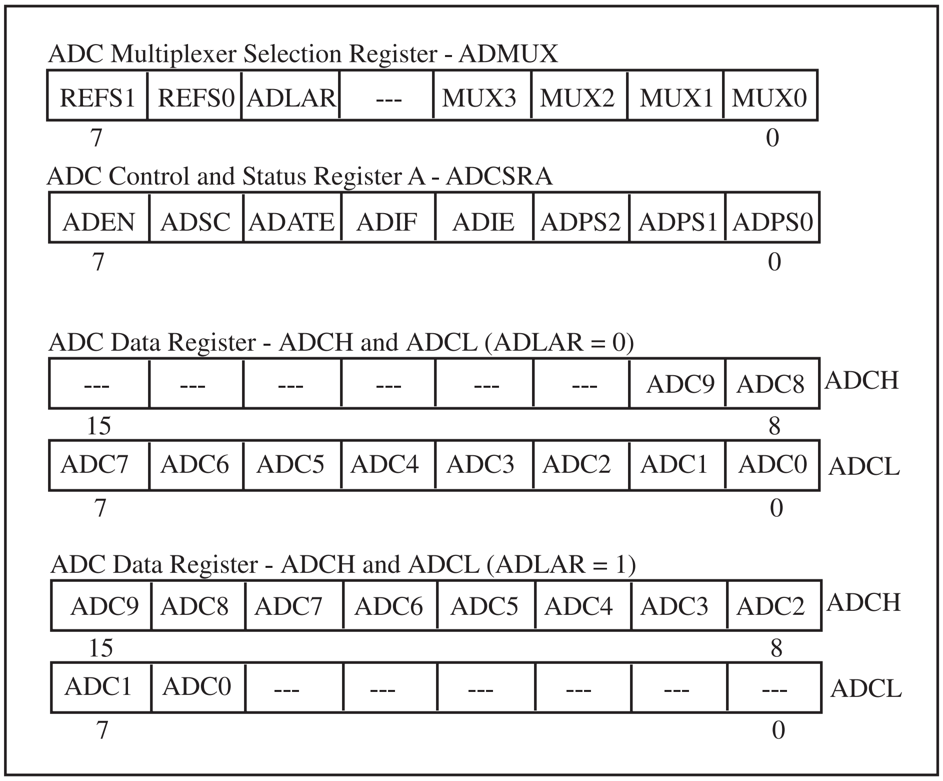

ADC Multiplexer Selection Register - ADMUX
REFS1 | REFS0 | ADLAR | --- | MUX3 | MUX2 | MUX1 | MUX0
7
0
ADC Control and Status Register A - ADCSRA
ADEN | ADSC | ADATE | ADIF | ADIE | ADPS2 | ADPS1 | ADPS0
7
0
ADC Data Register - ADCH and ADCL (ADLAR = 0)
--- | --- | --- | --- | --- | --- | ADC9 | ADC8
ADCH
15
8
ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADC1 | ADC0
ADCL
7
0
ADC Data Register - ADCH and ADCL (ADLAR = 1)
ADC9 | ADC8 | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2
ADCH
15
8
ADC1 | ADC0 | --- | --- | --- | --- | --- | ---
ADCL
7
0

Figure 5.9: ADC Registers. Adapted from Atmel.

- REFS[1:1] = 11: Internal 1.1 VDC voltage reference with an external capacitor at the AREF pin

5.5.2.2 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. 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

5.5.2.3 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.

5.6 PROGRAMMING THE ADC USING THE ARDUINO DEVELOPMENT ENVIRONMENT

The Arduino Development Environment 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 to 1023, corresponding to the voltage span from 0 to 5 VDC.

Recall that we introduced the use of this function in the Application section at the end of Chapter 3. The analogRead function was used to read the analog output values from the three IR sensors on the Blinky 602A robot.

5.7 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 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;
                                       //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 results
while (!(ADCSRA & 0x10));               //Check if conversion is ready
ADCSRA |= 0x10;                         //Clear Conv rdy flag - set the bit
binary_weighted_voltage_low = ADCL;     //Read 8 low bits first (important)
                                        //Read 2 high bits, multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage = binary_weighted_voltage_low |
                          binary_weighted_voltage_high;
return binary_weighted_voltage;         //ADCH:ADCL
}

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

5.8 EXAMPLE: ADC RAIN GAGE INDICATOR

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 analog-to-digital converter to illuminate up to eight LEDs based on the input voltage. A 10k trimmer potentiometer is connected to the ADC channel to vary the input voltage. We first provide a solution using the Arduino Development Environment with the Arduino Duemilanove processing board. Then a solution employing the ATmega328 programmed in C is provided.

5.8.1 ADC RAIN GAGE INDICATOR USING THE ARDUINO DEVELOPMENT ENVIRONMENT

The circuit configuration employing the Arduino Duemilanove processing board is provided in Figure 5.10. The DIGITAL pins of the microcontroller are used to communicate with the LED interface circuit. We describe the operation of the LED interface circuit in Chapter 8.

The sketch to implement the project requirements is provided below. As in previous examples, we define the Arduino Duemilanove 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 10K trimmer potentiometer and illuminates a series of LEDs corresponding to the sensed voltage levels.

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

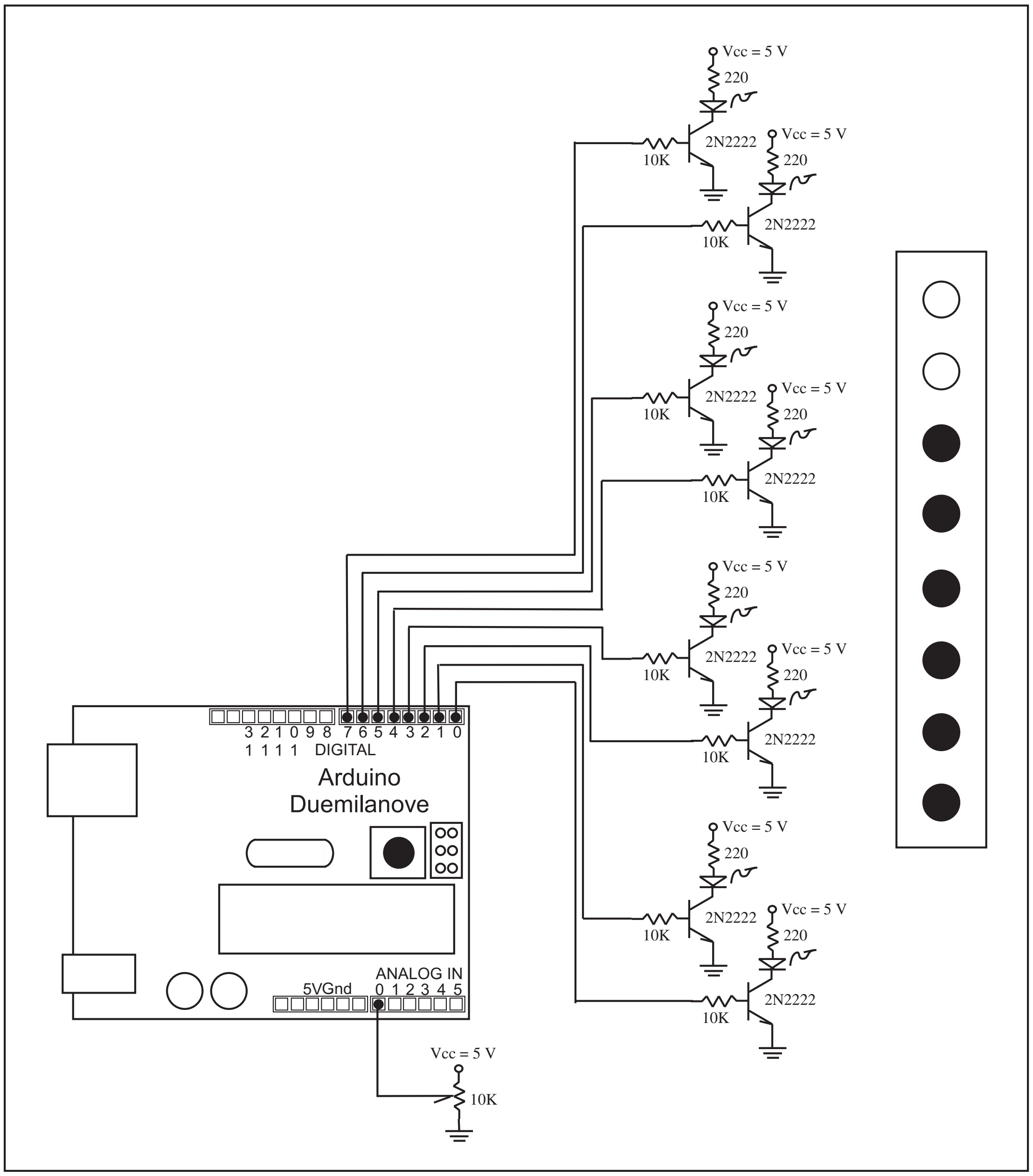


Figure 5.10: ADC with rain gage level indicator.

```
#define trim_pot   0                 //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
#define LED6       6                 //digital pin
#define LED7       7                 //digital pin


int trim_pot_reading;                //declare variable for trim pot

void setup()
  {
                                     //LED indicators - wall detectors
  pinMode(LED0, OUTPUT);             //configure pin 0 for digital output
  pinMode(LED1, OUTPUT);             //configure pin 1 for digital output
  pinMode(LED2, OUTPUT);             //configure pin 2 for digital output
  pinMode(LED3, OUTPUT);             //configure pin 3 for digital output
  pinMode(LED4, OUTPUT);             //configure pin 4 for digital output
  pinMode(LED5, OUTPUT);             //configure pin 5 for digital output
  pinMode(LED6, OUTPUT);             //configure pin 6 for digital output
  pinMode(LED7, OUTPUT);             //configure pin 7 for digital output
  }

void loop()
  {
                                     //read analog output 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
}

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

5.8.2 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 5.11. PORT D of the microcontroller is used to communicate with the LED interface circuit. In this example, we extend the requirements of the project:

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

The algorithm for the project was written by Anthony (Tony) Kunkel, MSEE and Geoff Luke, MSEE, 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.

```
//*************************************************************************
//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
//Fourth: Delay 30 ms when ever PORTD is updated
//*************************************************************************

//ATMEL register definitions for ATmega328
```

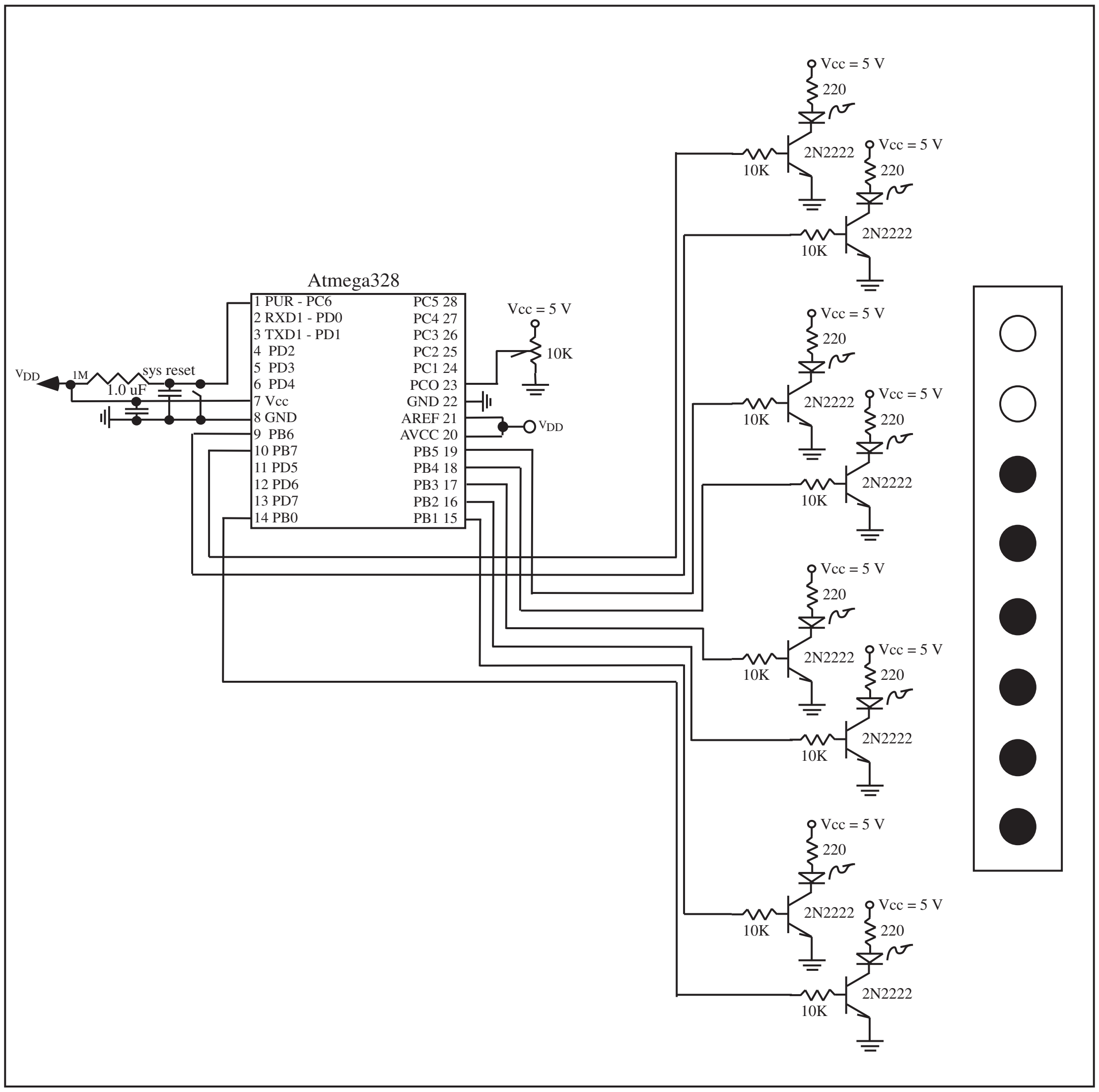


Figure 5.11: ADC with rain gage level indicator.

```
#include<iom328pv.h>

//function prototypes
void display_increment(void);        //Function
to display increment to PORTD
void display_decrement(void);        //Function
to display decrement to PORTD
void rain_gage(void)
void InitADC(void);                  //Initialize ADC converter
unsigned int ReadADC();              //Read specified ADC channel
void delay_30ms(void);               //Function to delay 30 ms

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

int main(void)
{
display_increment();                 //Display incrementing binary on

                                     //PORTD from 0-255
delay_30ms();                        //Delay 30 ms

display_decrement();                 //Display decrementing binary on

                                     //PORTD from 255-0
delay_30ms();                        //Delay 30 ms

InitADC();

while(1)
  {
  rain_gage();                       //Display gage info on PORTD
  delay_30ms();                      //Delay 30 ms
  }

return 0;
}
```

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

//*************************************************************************
//30 ms delay based on an 8MHz clock
//*************************************************************************

void delay_30ms(void)
{
int i, k;

for(i=0; i<400; i++)
  {
  for(k=0; k<300; k++)
      {
      asm("nop");             //assembly language 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_30ms();               //wait 30 ms
  }
}
```

```
//*************************************************************************
//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_30ms();               //wait 30 ms
  }
}


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

//*************************************************************************
//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;
  }
else if(ADCValue < 256)
  {
  PORTD = 0x03;
  }
else if(ADCValue < 384)
  {
  PORTD = 0x07;
  }
else if(ADCValue < 512)
  {
  PORTD = 0x0F;
  }
else if(ADCValue < 640)
  {
   PORTD = 0x1F;
  }
else if(ADCValue < 768)
  {
  PORTD = 0x3F;
  }
else if(ADCValue < 896)
  {
  PORTD = 0x7F;
  }
else
  {
  PORTD = 0xAA;
  }
}
//*************************************************************************
//*************************************************************************
```

5.8.3 ADC RAIN GAGE USING THE ARDUINO DEVELOPMENT ENVIRONMENT—REVISITED

If you carefully compare the two implementations of the rain gage indicator provided in the two previous examples, you will note that some activities are easier to perform in the Arduino Development

Environment while others are easier to accomplish in C. Is it possible to mix the two techniques for a more efficient sketch? The answer is "yes!"

In this example, we revise the earlier Arduino Development Environment sketch using portions of the original sketch with the ATmega328 control algorithm code to significantly shorten the program. In particular, we will address PORTD directly in the sketch to shorten up the code when the LEDs are illuminated. Recall in Chapter 1, we provided the open source schematic of the Arduino Duemilanove processing board (Figure ??). Careful study of the schematic reveals that DIGITAL pins 7 to 0 are connected to PORTD of the Arduino Duemilanove processing board. This allows direct configuration of PORTD using C language constructs.

```
//*************************************************************************
#define trim_pot   0                 //analog input pin

int trim_pot_reading;                //declare variable for trim pot

void setup()
  {
  DDRD = 0xFF;                       //Set PORTD (DIGITAL pins 7 to 0)
  }                                  //as output

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

   if(trim_pot_reading < 128)
     {
     PORTD = 0x01;                   //illuminate LED 0
     }
   else if(trim_pot_reading < 256)
     {
     PORTD = 0x03;                   //illuminate LED 0-1
     }
   else if(trim_pot_reading < 384)
     {
     PORTD = 0x07;                   //illuminate LED 0-2
     }
   else if(trim_pot_reading < 512)
     {
     PORTD = 0x0F;                   //illuminate LED 0-3
```

```
      }
    else if(trim_pot_reading < 640)
      {
      PORTD = 0x1F;                    //illuminate LED 0-4
      }
  else if(trim_pot_reading < 768)
    {
    PORTD = 0x3F;                      //illuminate LED 0-5
    }
  else if(trim_pot_reading < 896)
    {
    PORTD = 0x7F;                      //illuminate LED 0-6
    }
  else
    {
    PORTD = 0xFF;                      //illuminate LED 0-7
    }
delay(500);                            //delay 500 ms
}

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

5.9 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 +Vcc as shown in Figure 5.12a). When the input signal falls below the threshold the op amp will saturate at a voltage slightly greater than –Vcc. If a single-sided op amp is used in the circuit (e.g., LM324), the –Vcc 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 5.12c). This provides a flash converter type ADC. It is a hardware version of a rain gage indicator. In Chapter 1, we provided a 14-channel version for use in a laboratory instrumentation project.

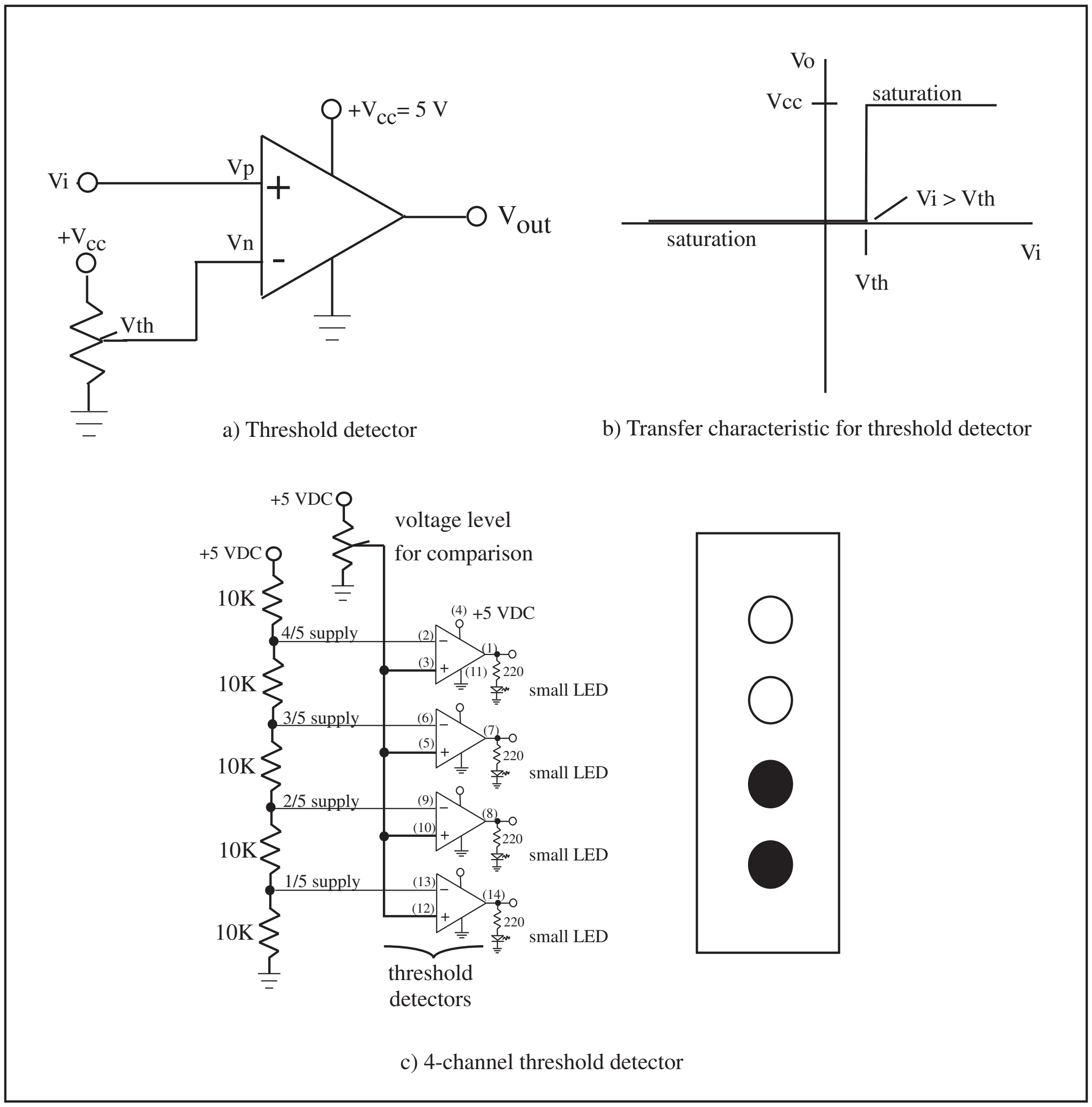


Figure 5.12: One-bit ADC threshold detector.

5.10 DIGITAL-TO-ANALOG CONVERSION (DAC)

Once a signal is acquired to a digital system with the help of the analog-to digital conversion 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 5.13.

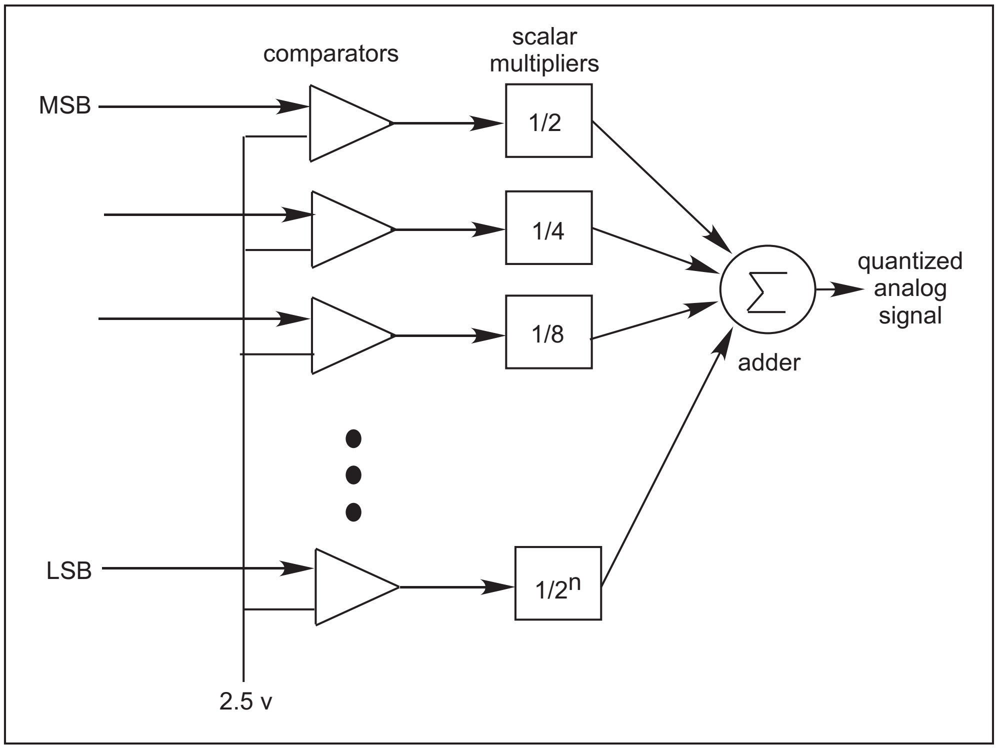


Figure 5.13: A summation method to convert a digital signal into a quantized analog signal. Comparators are used to clean up incoming signals and the resulting values are multiplied by a scalar multiplier and the results are added to generate the output signal. For the final analog signal, the quantized analog signal should be connected to a low pass filter followed by a transducer interface circuit.

With the summation method of digital-to-analog conversion, a digital signal, represented by a set of ones and zeros, enters the digital-to-analog converter 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 5.13. 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 digital-to-analog converter to the input of an output transducer.

5.10.1 DAC WITH THE ARDUINO DEVELOPMENT ENVIRONMENT

The analogWrite command within the Arduino Development Environment issues a signal from 0 to 5 VDC by sending a constant from 0 to 255 using pulse width modulation (PWM) techniques. This signal, when properly filtered, serves as a DC signal. The Blinky 602A control sketch provided in Chapter 3 used the analogWrite command to issue drive signals to the robot motors to navigate the robot through the maze.

The form of the analogWrite command is the following:
analogWrite(output pin, value);

5.10.2 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)
- 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)
- 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, but we will examine the TLC5628 in more detail.

5.10.3 OCTAL CHANNEL, 8-BIT DAC VIA THE SPI

The Texas Instruments (TI) TLC5628 is an eight channel, 8-bit DAC connected to the microcontroller via the SPI. Reference Figure 5.14a). It has a wide range of features packed into a 16-pin chip including [Texas Instrument]:

- Eight individual, 8-bit DAC channels,
- Operation from a single 5 VDC supply,
- Data load via the SPI interface,
- Programmable gain (1X or 2X), and

- A 12-bit command word to program each DAC.

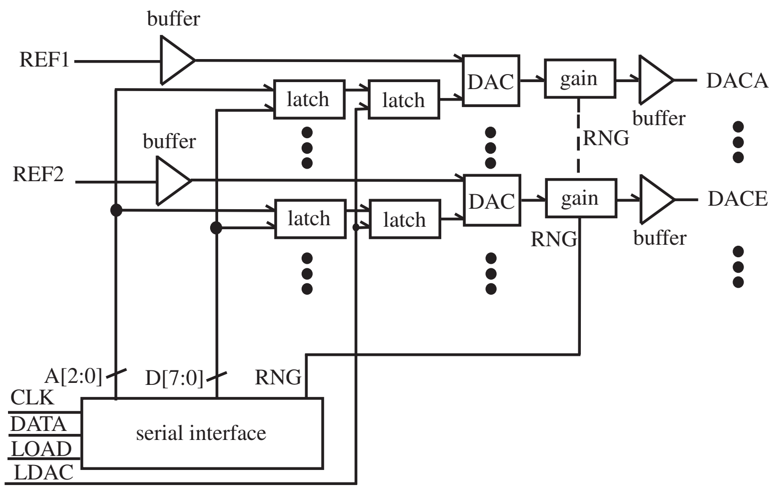
Figure 5.14b) provides the interconnection between the ATmega328 and the TLC5628. The ATmega328 SPI's SCK line is connected to the TLC5628: the MOSI to the serial data line and PORTB[2] to the Load line. As can be seen in the timing diagram, two sequential bytes are sent from the ATmega328 to select the appropriate DAC channel and to provide updated data to the DAC. The Load line is pulsed low to update the DAC. In this configuration, the LDAC line is tied low. The function to transmit data to the DAC is left as an assignment for the reader at the end of the chapter.

5.11 APPLICATION: ART PIECE ILLUMINATION SYSTEM – REVISITED

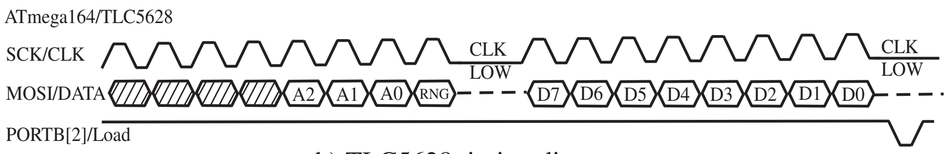
In Chapter 2, we investigated an illumination system for a painting. The painting was to be illuminated via high intensity white LEDs. The LEDs could be mounted in front of or behind the painting as the artist desired. We equipped the lighting system with an IR sensor to detect the presence of someone viewing the piece. We also wanted to adjust the intensity of the lighting based on how the close viewer was to the art piece. A circuit diagram of the system is provided in Figure 5.15.

The Arduino Development Environment sketch to sense how away the viewer is and issue a proportional intensity control signal to illuminate the LED is provided below. The analogRead function is used to obtain the signal from the IR sensor. The analogWrite function is used to issue a proportional signal.

```
//*********************************************************************
                                          //analog input pins
#define viewer_sensor       0             //analog pin - left IR sensor

                                          //digital output pins
#define illumination_output  0            //illumination output pin

unsigned int viewer_sensor_reading;       //current value of sensor output

void setup()
  {
  pinMode(illumination_output, OUTPUT);   //config. pin 0 for dig. output
  }

void loop()
  {                                       //read analog output from
                                          //IR sensors
   viewer_sensor_reading  = analogRead(viewer_sensor);
```



a) TLC5628 octal 8-bit DACs



b) TLC5628 timing diagram

A[2:0]	DAC Updated
000	DACA
001	DACB
010	DACC
011	DACD
100	DACE
101	DACF
110	DACG
111	DACH

D[7:0]	Output Voltage
0000_0000	GND
0000_0001	(1/256) x REF(1+RNG)
:	:
:	:
:	:
:	:
:	:
1111_1111	(255/256) x REF (1+RNG)

c) TLC5628 bit assignments.

Figure 5.14: Eight channel DAC. Adapted from [Texas Instrument].

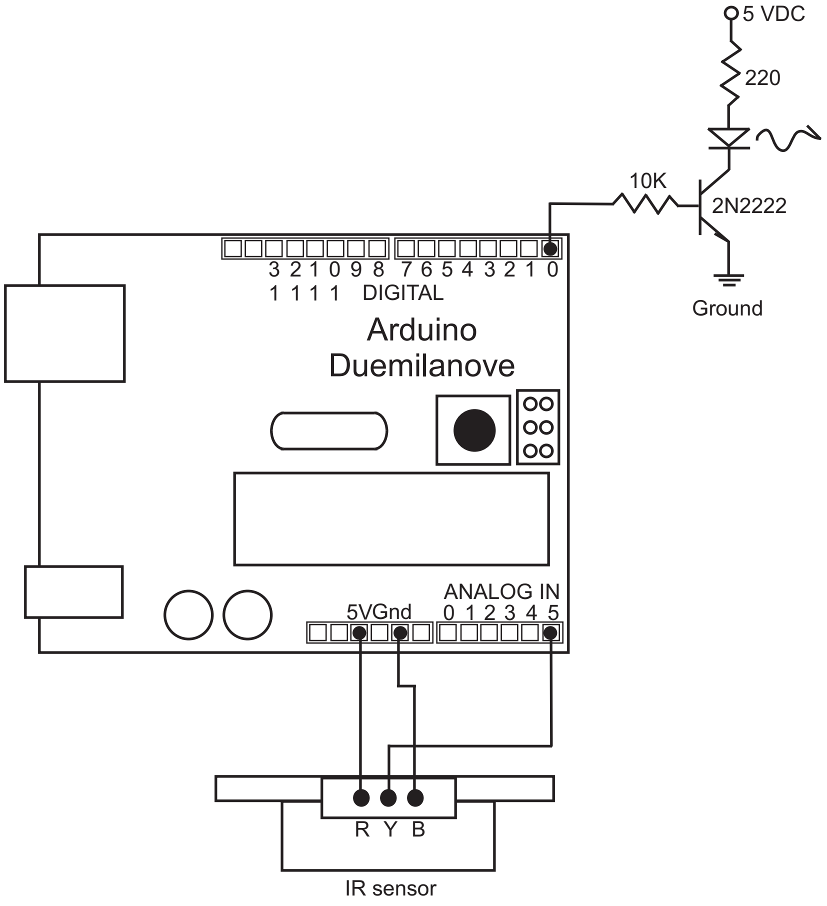


Figure 5.15: IR sensor interface.

```
if(viewer_sensor_reading < 128)
  {
  analogWrite(illumination_output,31);  //0 (off) to 255 (full speed)
  }
else if(viewer_sensor_reading < 256)
  {
  analogWrite(illumination_output,63);  //0 (off) to 255 (full speed)
  }
else if(viewer_sensor_reading < 384)
```

```
      {
      analogWrite(illumination_output,95);  //0 (off) to 255 (full speed)
      }
    else if(viewer_sensor_reading < 512)
      {
      analogWrite(illumination_output,127); //0 (off) to 255 (full speed)
      }
    else if(viewer_sensor_reading < 640)
      {
      analogWrite(illumination_output,159); //0 (off) to 255 (full speed)
      }
    else if(viewer_sensor_reading < 768)
      {
      analogWrite(illumination_output,191); //0 (off) to 255 (full speed)
      }
    else if(viewer_sensor_reading < 896)
      {
      analogWrite(illumination_output,223); //0 (off) to 255 (full speed)
      }
    else
      {
      analogWrite(illumination_output,255); //0 (off) to 255 (full speed)
      }
delay(500);                                 //delay 500 ms
}

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

5.12 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 analog to digital converters: sampling, quantization, and encoding. We also presented the quantization errors and the data rate associated with the ADC process. The dynamic range of an analog-to-digital converter, 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 ADC process and an implementation using a multi-channel DAC connected to the ATmega328 SPI system.

5.13 REFERENCES

- *Atmel 8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash, ATmega328, ATmega328L,* data sheet: 2466L-AVR-06/05, Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131.
- Barrett S, Pack D (2006) Microcontrollers Fundamentals for Engineers and Scientists. Morgan and Claypool Publishers. DOI: 10.2200/S00025ED1V01Y200605DCS001
- Barrett S and Pack D (2008) Atmel AVR Microcontroller Primer Programming and Interfacing. Morgan and Claypool Publishers. DOI: 10.2200/S00100ED1V01Y200712DCS015
- Barrett S (2010) Embedded Systems Design with the Atmel AVR Microcontroller. Morgan and Claypool Publishers. DOI: 10.2200/S00225ED1V01Y200910DCS025
- Roland Thomas and Albert Rosa, *The Analysis and Design of Linear Circuits, Fourth Edition,* Wiley & Sons, Inc., New York, 2003.
- M.A. Hollander, editor, *Electrical Signals and Systems, Fourth Edition,* McGraw-Hill Companies, Inc, 1999.
- Daniel Pack and Steven Barrett, *Microcontroller Theory and Applications: HC12 and S12,* Prentice Hall, 2ed, Upper Saddle River, New Jersey 07458, 2008.
- Alan Oppenheim and Ronald Schafer, *Discrete-time Signal Processing, Second Edition,* Prentice Hall, Upper Saddle River, New Jersey, 1999.
- John Enderle, Susan Blanchard, and Joseph Bronzino, *Introduction to Biomedical Engineering,* Academic Press, 2000.
- L. Faulkenberry, *An Introduction to Operational Amplifiers,* John Wiley & Sons, New York, 1977.
- P. Horowitz and W. Hill, *The Art of Electronics,* Cambridge University Press, 1989.
- L. Faulkenberry, *Introduction to Operational Amplifiers with Linear Integrated Circuit Applications*, 1982.
- D. Stout and M. Kaufman, *Handbook of Operational Amplifier Circuit Design* McGraw-Hill Book Company, 1976.
- S. Franco, *Design with Operational Amplifiers and Analog Integrated Circuits, third edition,* McGraw-Hill Book Company, 2002.
- *TLC5628C, TLC5628I Octal 8-bit Digital-to-Analog Converters,* Texas Instruments, Dallas, TX, 1997.

5.14 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, the quantization bits, and the technique used for the conversion.

5. A flex sensor provides 10K ohm of resistance for 0 degrees flexure and 40K ohm of resistance for 90 degrees 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 5.4.

8. If a resistor was connected between the non-inverting terminal and ground in the inverting amplifier configuration of Figure 5.4a), 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?

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 Duemilanove using built-in features of the Arduino Development Environment.
- Use the interrupt system to implement a real time clock.
- 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.

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 in detail. We provide several examples on how to program an interrupt in C and also using the built-in features of the Arduino Development Environment.

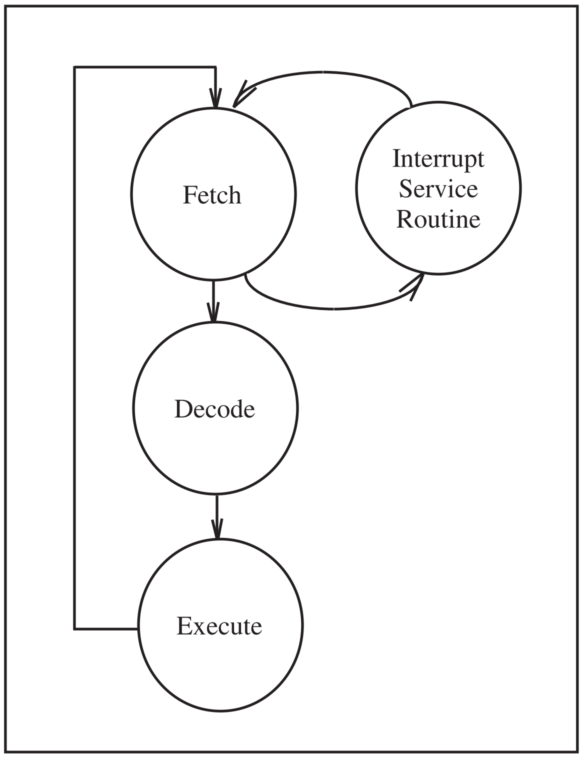


Figure 6.1: Microcontroller Interrupt Response.

6.2 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.

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 for that interrupt into the program counter. The interrupt service routine will then commence. Execution of

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: Atmel AVR ATmega328 Interrupts. (Adapted from figure used with permission of Atmel, Incorporated.)

the ISR continues until the return from interrupt instruction (reti) is encountered. Program control then reverts back to the main program.

6.3 INTERRUPT PROGRAMMING

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 next two examples that follow, we illustrate how to accomplish these steps. We use the ImageCraft ICC AVR compiler which contains excellent support for interrupts. Other compilers have similar features.

6.4 PROGRAMMING INTERRUPTS IN C AND THE ARDUINO DEVELOPMENT ENVIRONMENT

In this section, we provide two 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 and also with the Arduino Development Environment built-in features. For the C examples, we use the ImageCraft ICC AVR compiler.

The ImageCraft ICC AVR compiler uses the following syntax to link an interrupt service routine to the correct interrupt vector address:

```
#pragma interrupt_handler timer_handler:4

void timer_handler(void)
{
:
:
}
```

As you can see, 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 Figure 6.2. The ISR is then written like any other function. It is important that the ISR name used in the #pragma instruction identically matches the name of the ISR in the function body. Since the compiler knows the function is an ISR, it will automatically place the assembly language RETI instruction at the end of the ISR.

6.4.1 EXTERNAL INTERRUPT PROGRAMMING

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.

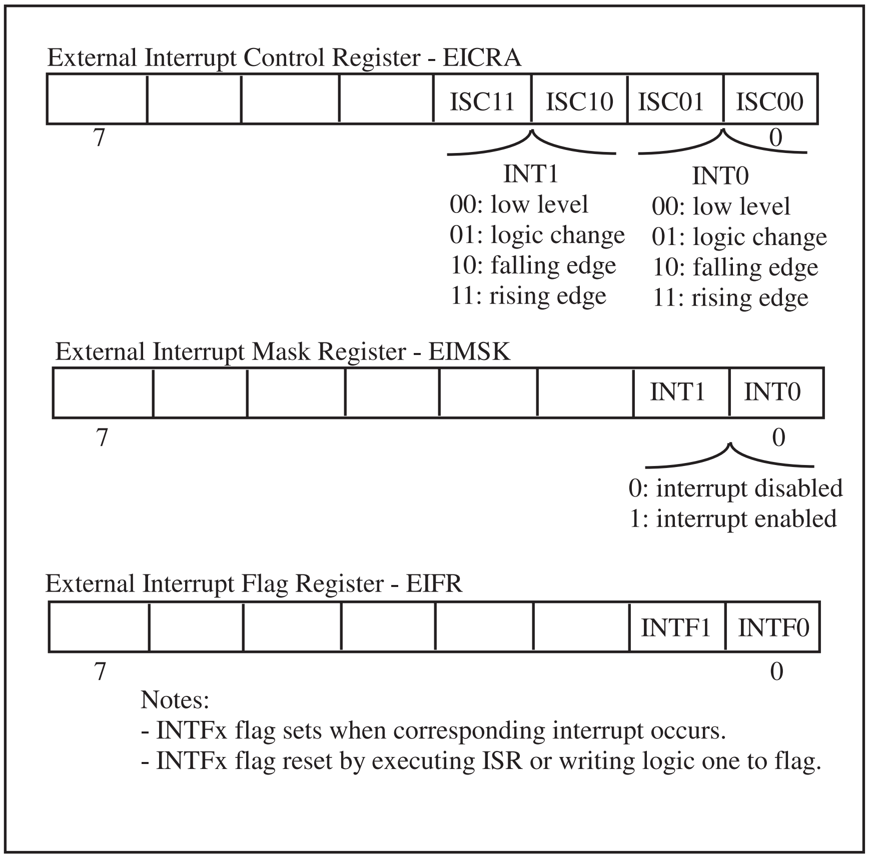

External Interrupt Control Register - EICRA
ISC11
ISC10
ISC01
ISC00
7
0
INT1
00: low level
01: logic change
10: falling edge
11: rising edge
INT0
00: low level
01: logic change
10: falling edge
11: rising edge
External Interrupt Mask Register - EIMSK
INT1
INT0
7
0
0: interrupt disabled
1: interrupt enabled
External Interrupt Flag Register - EIFR
INTF1
INTF0
7
0
Notes:
- INTFx flag sets when corresponding interrupt occurs.
- INTFx flag reset by executing ISR or writing logic one to flag.

Figure 6.3: Interrupt INT0 and INT1 Registers.

6.4.1.1 Programming external interrupts in C

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();
:



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

//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 for 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).

6.4.1.2 Programming external interrupts using the Arduino Development Environment built-in features

The Arduino Development Environment 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 Duemilanove 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 Arduino Development Environment 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.

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

6.4.2 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.

6.4.2.1 Programming an internal interrupt in C

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 microseconds. 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 specified number 6.55ms
void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);  //initialize timer0 overf.interrupt


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



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

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

void main(void)
{
init_timer0_ovf_interrupt();           //initialize 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;                       //divide timer0 timebase by 256,
                                     //overflow occurs every 6.55ms
TIMSK0 = 0x01;                       //enable timer0 overflow interrupt
asm("SEI");                          //enable global interrupt
}

//*********************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared 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 spec. number of interpts.
  }
}

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

6.4.2.2 Programming an internal interrupt using the Arduino Development Environment

The Arduino Development Environment 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 Arduino Development Environment using AVR-GCC conventions. Also, the timing functions in the previous example assumed a time base of 10 MHz. The Arduino Duemilanove 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 number of Timer/Counter0
                                    //Overflow interrupts

void setup()
{
init_timer0_ovf_interrupt();        //initialize Timer/Counter0 Overfl.
}

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; //divide timer0 timebase by 256, overflow occurs every 4.1 ms
TIMSK0 = 0x01; //enable 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 for spec. number of intrpts.
  }
}

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

6.5 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 pulse width modulation (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 foreground while monitoring safety related status in the background using interrupts as illustrated in Figure 6.4.

6.6 INTERRUPT EXAMPLES

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

6.6.1 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

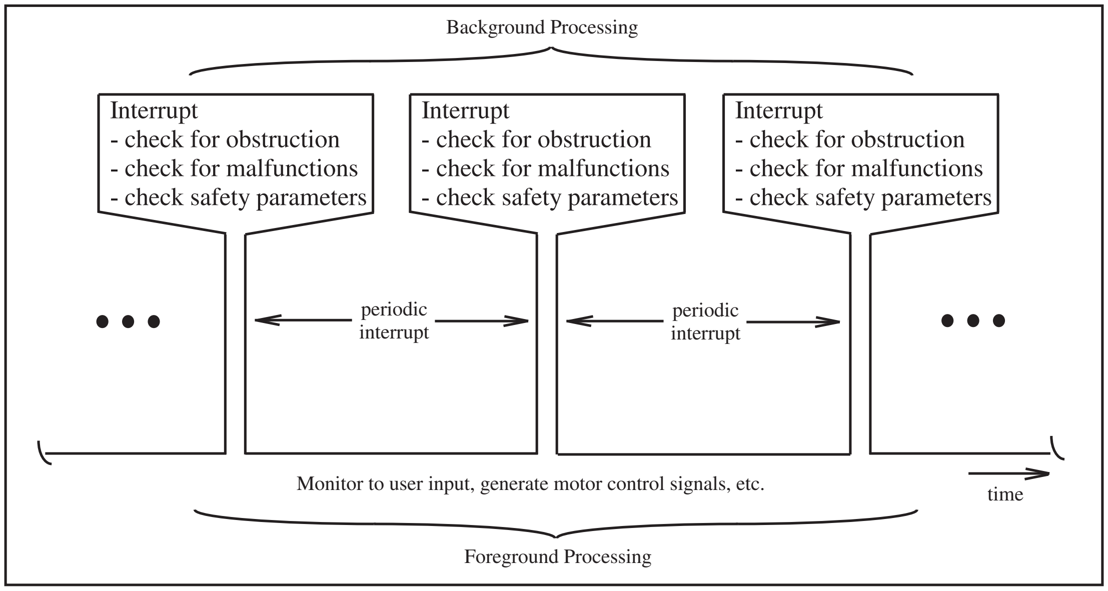


Figure 6.4: Interrupt used for background processing. The microcontroller responds to user input status in the foreground while monitoring safety related status in the background using interrupts.

be displayed on a liquid crystal display. This is also a useful technique for generating very long delays in a microcontroller.

```
//function prototypes******************************************************
                                    //delay specified number 6.55ms

void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);//initialize timer0 overflow interrupt



//interrupt handler definition*********************************************
                                    //interrupt 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();         //initialize 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; //divide timer0 timebase by 256, overfl. occurs every 6.55ms

TIMSK0 = 0x01; //enable timer0 overflow interrupt
asm("SEI");    //enable global interrupt
}

//*************************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing the
//corresponding interrupt handling vector.
//*************************************************************************

void timer0_interrupt_isr(void)
```

```
{

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

                                           //Update second counter
if(ms_ctr == 154)                          //counter equates to 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
  }
}

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

6.6.2 REAL TIME CLOCK USING THE ARDUINO DEVELOPMENT ENVIRONMENT

In this example, we reconfigure the previous example using the Arduino Development Environment. The timing functions in the previous example assumed a time base of 10 MHz. The Arduino Duemilanove 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
                                       //counter equates to 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, overfl. occurs every 4.1 ms
TIMSK0 = 0x01; //enable timer0 overflow interrupt
asm("SEI");    //enable global interrupt
}
//*************************************************************************
```

6.6.3 INTERRUPT DRIVEN USART IN C

In Chapter 4, 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 6.5. The actual encoder is new technology and production models are not available for evaluation.

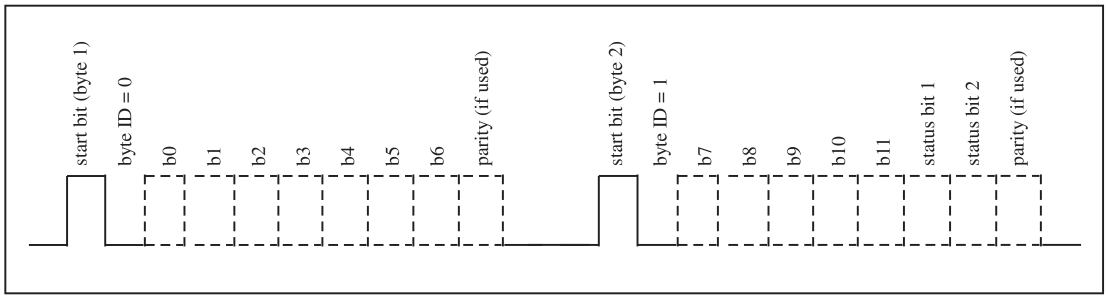


Figure 6.5: Encoder data format. The position data is sent serially at 9600 Baud as two sequential bytes.

Since the actual encoder is not available for evaluation, another Atmel ATmega328 will be used to send signals in identical format and Baud rate as the encoder. The test configuration is illustrated in Figure 6.6. 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 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 6.5. 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.

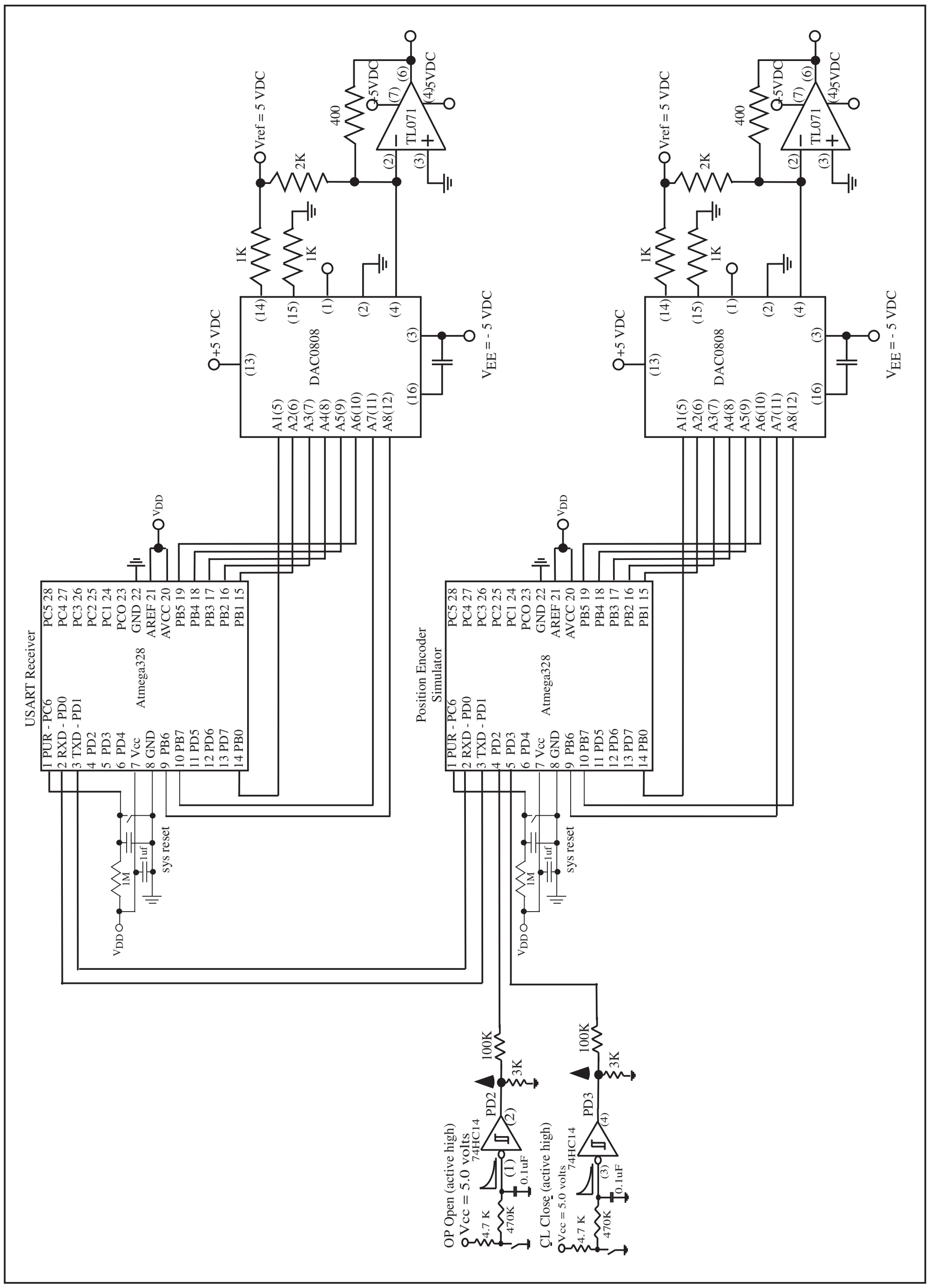


Figure 6.6: Encoder test configuration.

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: April 15, 2010
//file: encode.c
//target controller: ATMEL ATmega328
//
//ATmega328 clock source: internal 8 MHz clock
//
//ATMEL 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>

//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 configuration 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: April 15, 2010
//file: receive.c
//target controller: ATMEL ATmega328
//
//ATmega328 clock source: internal 8 MHz clock
//
//ATMEL 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 definition
#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 configuration to deflt.
USART_init();
                                       //limited startup features
                                       //main activity loop - processor will
                                       //continually cycle through 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 bits
                               //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 to logic 1
  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;
}

//*************************************************************************
//end of file
//*************************************************************************
```

6.7 SUMMARY

In this chapter, we provided an introduction to the interrupt features available aboard the ATmega328 and the Arduino Duemilanove 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.

6.8 REFERENCES

- *Atmel 8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash, ATmega48PA, 88PA, 168PA, 328P* data sheet: 8161D-AVR-10/09, Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131.
- Barrett S, Pack D (2006) Microcontrollers Fundamentals for Engineers and Scientists. Morgan and Claypool Publishers. DOI: 10.2200/S00025ED1V01Y200605DCS001
- Barrett S and Pack D (2008) Atmel AVR Microcontroller Primer Programming and Interfacing. Morgan and Claypool Publishers. DOI: 10.2200/S00100ED1V01Y200712DCS015
- Barrett S (2010) Embedded Systems Design with the Atmel AVR Microcontroller. Morgan and Claypool Publishers. DOI: 10.2200/S00225ED1V01Y200910DCS025

6.9 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 hour 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 hour timer example to generate an active high logic pulse on a microcontroller pin of your choice for three seconds. 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
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 Atmel ATmega328.
- Describe the four operating modes of the Atmel ATmega328 timing system.
- Describe the register configurations for the ATmega328's Timer 0, Timer 1, and Timer 2.
- Program the Arduino Duemilanove using the built-in timer features of the Arduino Development Environment.
- Program the ATmega328 timer system for a variety of applications using C.

7.1 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 Atmel ATmega328 microcontroller to perform time related functions. 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: Timer 0, Timer 1, and Timer 2 and their different modes of operation. We then review the built-in timing functions of the Arduino Development Environment and conclude the chapter with a variety of examples.

7.2 TIMING RELATED TERMINOLOGY

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

7.2.1 FREQUENCY

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

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

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

7.2.2 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 second. To find a period, given the frequency of a signal, or vice versa, we simply need to remember their inverse relationship $f = \frac{1}{T}$ where f and T represent a frequency and the corresponding period, respectively. Both periods and frequencies of signals 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 milliseconds. The constraint then forces the design team that monitors the slippage to program their monitoring system to check a slippage at a rate of 50 Hz.

7.2.3 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 7.1(a) is on for 50 percent of the signal period and off for the rest of the period. The pulse shown in (b) is on for only 25 percent of the same period as the signal in (a) and off for 75 percent of the period. The duty cycle is defined as the percentage of the period a signal is on or logic high. Therefore, we call the signal in Figure 7.1(a) as a periodic pulse signal with a 50 percent duty cycle and the corresponding signal in (b), a periodic pulse signal with a 25 percent duty cycle.

7.3 TIMING SYSTEM OVERVIEW

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

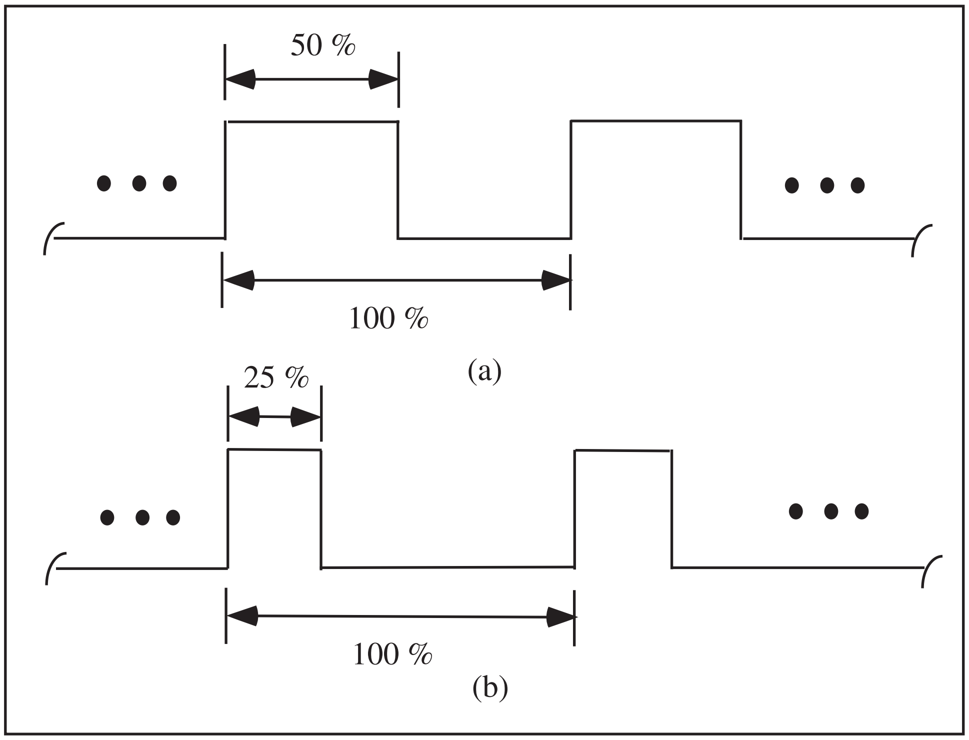


Figure 7.1: Two signals with the same period but different duty cycles. The top figure (a) shows a periodic signal with a 50% duty cycle and the lower figure (b) displays a periodic signal with a 25% duty cycle.

count up (increment) each time it sees a rising edge (or a falling edge) of a clock signal. Thus, if a clock is running at the rate of 2 MHz, the free running counter will count up or increment each 0.5 microseconds. All other timer related units reference the contents of the free running counter to perform input and output time related activities: measurement of time periods, capture of timing events, and generation of time related signals.

The 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 an internal fixed clock operating frequency of 1, 2, 4 or 8 MHz.

To provide for a wider range of frequency selections an external time source may be used. The external time sources, in order of increasing accuracy and stability, are 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. As previously mentioned, the maximum operating frequency of the ATmega328P with a 5 VDC supply voltage is 20 MHz. The Arduino Duemilanove processing board is equipped with a 16 MHz crystal oscillator time base.

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 a free running counter to capture external event times. We can use such ability to measure the period of an incoming signal, the width of a pulse, and the time of a signal logic change.

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.

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 sec and 1.6 sec, respectively. We can use these values to easily compute the pulse width of 0.1 second.

The second overall goal of the timer system is to generate signals to control external devices. Again an event simply means a change of logic states on an output pin of a microcontroller at a specified time. Now consider Figure 7.2. Suppose an external device connected to the microcontroller requires a pulse signal to turn itself on. Suppose the particular pulse the external device needs is 2 millisecond wide. In such situations, we can use the free running counter value to synchronize the time of desired logic state changes. Naturally, extending the same capability, we can also generate a periodic pulse with a fixed duty cycle or a varying duty cycle.

From the examples we discussed above, you may have wondered how a microcontroller can be used to compute absolute times from the relative free running counter values, say 1.5 second and 1.6 second. The simple answer is that we can not 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 microsecond and it takes approximately 32.78 milliseconds to count from 0 to 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, [1] can you find the pulse width when the free running counter is counting at 2 MHz? Let's first convert the two values into their corresponding decimal values, 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 $260 \times 0.5\, microseconds = 130\, microseconds$.

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

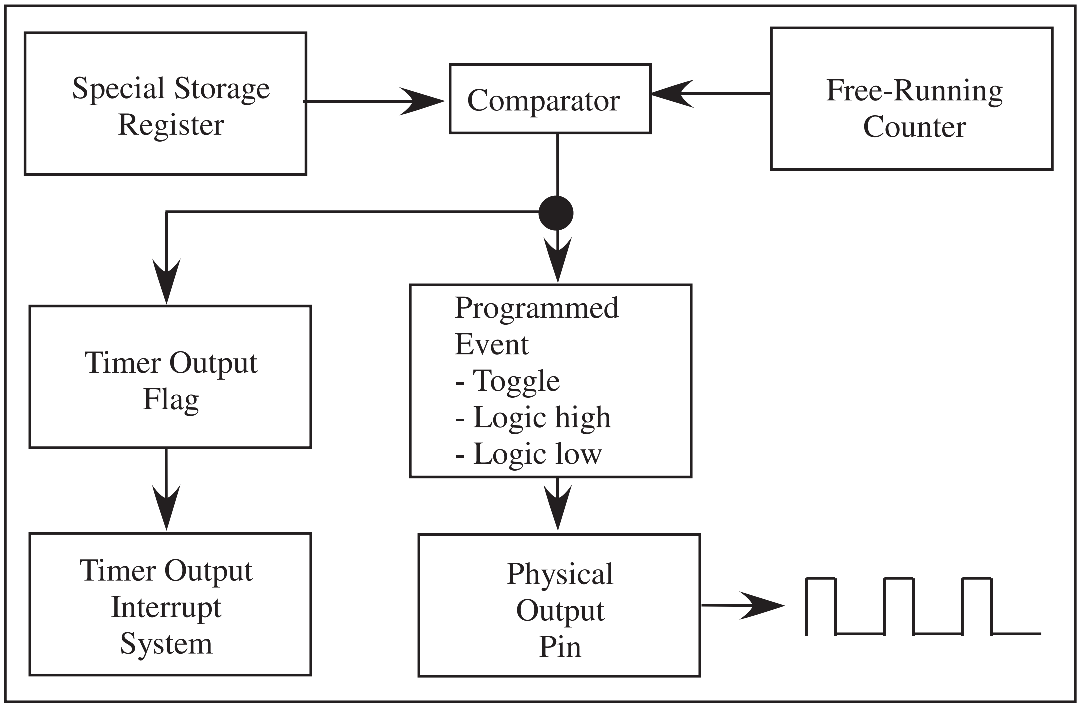


Figure 7.2: A diagram of a timer output system.

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.

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:

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

$$elapsed\ time = (elapsed\ clock\ ticks) \times (FRC\ clock\ period)\ [seconds]$$

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.

7.4 APPLICATIONS

In this section, we consider 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.

7.4.1 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 7.3 shows an incoming periodic signal to our microcontroller.

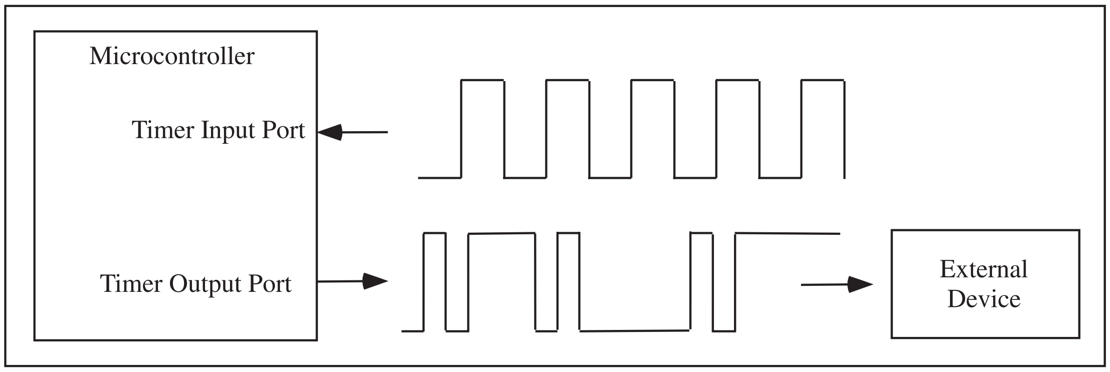


Figure 7.3: Use of the timer input and output systems of a microcontroller. The signal on top is fed into a timer input port. The captured signal is subsequently used to compute the input signal frequency. The signal on the bottom is generated using the timer output system. The signal is used to control an external device.

The first necessary step for the current task 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.

Now that the input timer system is configured appropriately, you now 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 microcontroller manufacturers have developed 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 \$1000 and \$5000, respectively. Note that these values are not absolute times but the representations of times reflected as the values of the free running counter. The period of the signal is \$4000 or 16384 in a decimal form. If we assume that the timer clock runs at 10 MHz, the period of the signal is 1.6384 msec, and the corresponding frequency of the signal is approximately 610.35 Hz.

7.4.2 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 a timer and configuring a physical input port pin are the same. In this application, however, the programmed event should be any logic state changes instead of looking for a rising or a falling edge as we have done in the previous section. If the polling technique is used, at each event detection, the corresponding flag must be cleared and a counter must be updated. If the interrupt technique is used, one must write an interrupt service routine within which the flag is cleared and a counter is updated.

7.4.3 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 subsection 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 7.3 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 7.3, one must compute the time required between the rising and the falling edges. Suppose that the external device requires a pulse which is 2 milliseconds 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 5.2, 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 7.3, let's assume that the main clock runs at 2 MHz, the free running counter is a 16 bit counter, and the name of the special register (16 bit register) where we can put appropriate values is output timer register. To generate the desired pulse, we can 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.

7.4.4 INDUSTRIAL IMPLEMENTATION CASE STUDY (PWM)

In this section, we discuss a well-known method to control the speed of a DC motor using a pulse width modulated (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 can not 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 7.4. As shown in

[hbt]

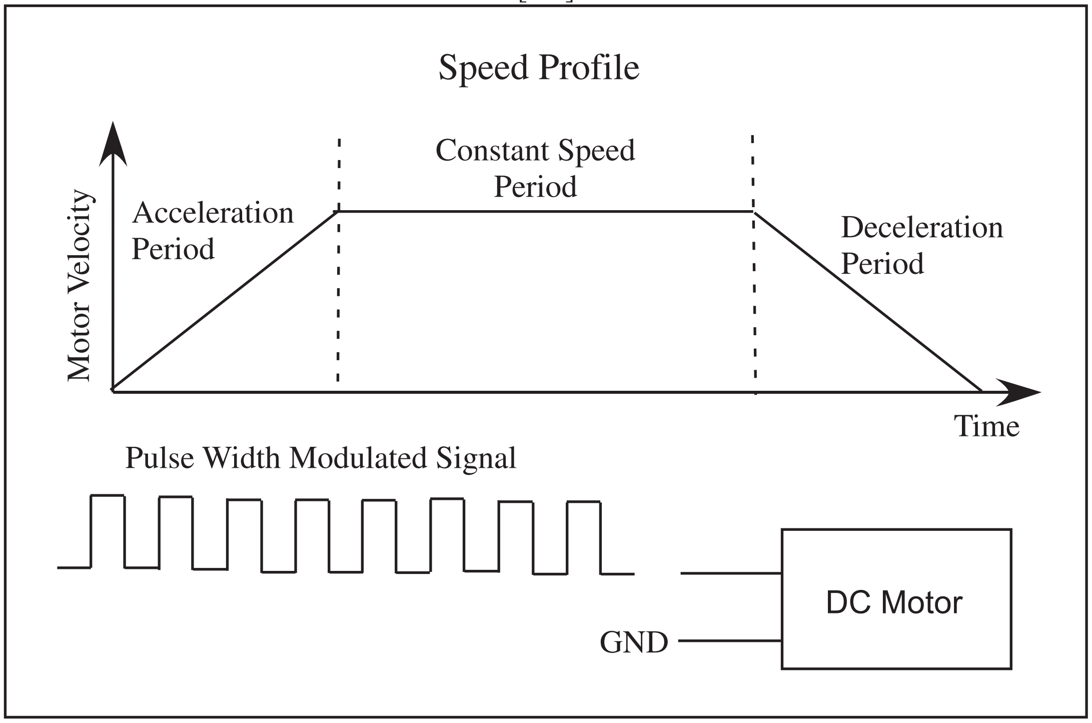


Figure 7.4: The figure shows the speed profile of a DC motor over time when a pulse-width-modulated signal is applied to the motor.

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 subsection. 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 $4000 to the register, provided that we are using a 16 bit free running counter. Once the falling edge is generated, we now need to go back and change the event to be a rising edge and the contents of the output timer counter value back to $0000. If we want to continue to generate a 25% duty cycle signal, then we must repeat the process indefinitely. Note that we are using the time for a free running counter to count from \$0000 to \$FFFF as one period.

Now suppose we want to increase the duty cycle to 50% over 1 sec and that the clock is running at 2 MHz. This means that the free running counter counts from \$0000 to \$FFFF every 32.768 milliseconds, and the free running counter will count from \$0000 to \$FFFF approximately 30.51 times over the period of one second. That is we need to increase the pulse width from \$4000 to \$8000 in approximately 30 turns, or approximately 546 clock counts every turn. This technique may be used to generate any desired duty cycle.

7.5 OVERVIEW OF THE ATMEL ATMEGA328 TIMER SYSTEM

The Atmel ATmega328 is equipped with a flexible and powerful three channel timing system. 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 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 two channels.

The features of the timing system are summarized in Figure 7.5. Timer 0 and 2 are 8-bit timers; whereas, Timer 1 is a 16-bit timer. 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 using the output compare channels. Additionally, Timer 1 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.

7.6 TIMER 0 SYSTEM

In this section, we discuss the features, overall architecture, modes of operation, registers, and programming of Timer 0. This information may be readily adapted to Timer 1 and Timer 2.

A Timer 0 block diagram is shown in Figure 7.6. The clock source for Timer 0 is provided via an external clock source at the T0 pin (PD4) 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).

Timer 0	**Timer 1**	**Timer 2**
- 8-bit timer/counter	- 16-bit timer/counter	- 8-bit timer/counter
- 10-bit clock prescaler	- 10-bit clock prescaler	- 10-bit clock prescaler
- Functions:	- Functions:	- Functions:
-- Pulse width modulation	-- Pulse width modulation	-- Pulse width modulation
-- Frequency generation	-- Frequency generation	-- Frequency generation
-- Event counter	-- Event counter	-- Event counter
-- Output compare -- 2 ch	-- Output compare -- 2 ch	-- Output compare -- 2 ch
- Modes of operation:	-- Input capture	- Modes of operation:
-- Normal	- Modes of operation:	-- Normal
-- Clear timer on	-- Normal	-- Clear timer on
compare match (CTC)	-- Clear timer on	compare match (CTC)
-- Fast PWM	compare match (CTC)	-- Fast PWM
-- Phase correct PWM	-- Fast PWM	-- Phase correct PWM
	-- Phase correct PWM	

Figure 7.5: Atmel timer system overview.

The TCCR0A register contains the WGM0[1:0] bits and the COM0A[1:0] (and B) bits. Whereas, the TCCR0B register contains the WGM0[2] bit. These bits are used to select the mode of operation for Timer 0 as well as tailor waveform generation for a specific application.

The timer clock source (clk_{Tn}) is fed to the 8-bit Timer/Counter Register (TCNT0). This register is incremented (or decremented) on each clk_{Tn} clock pulse. Timer 0 is also equipped with two 8-bit comparators that constantly compares the numerical content of TCNT0 to the Output Compare Register A (OCR0A) and Output Compare Register B (OCR0B). The compare signal from the 8-bit comparator is fed to the waveform generators. The waveform generators have a number of inputs to perform different operations with the timer system.

The BOTTOM signal for the waveform generation and the control logic, shown in Figure 7.6, 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.

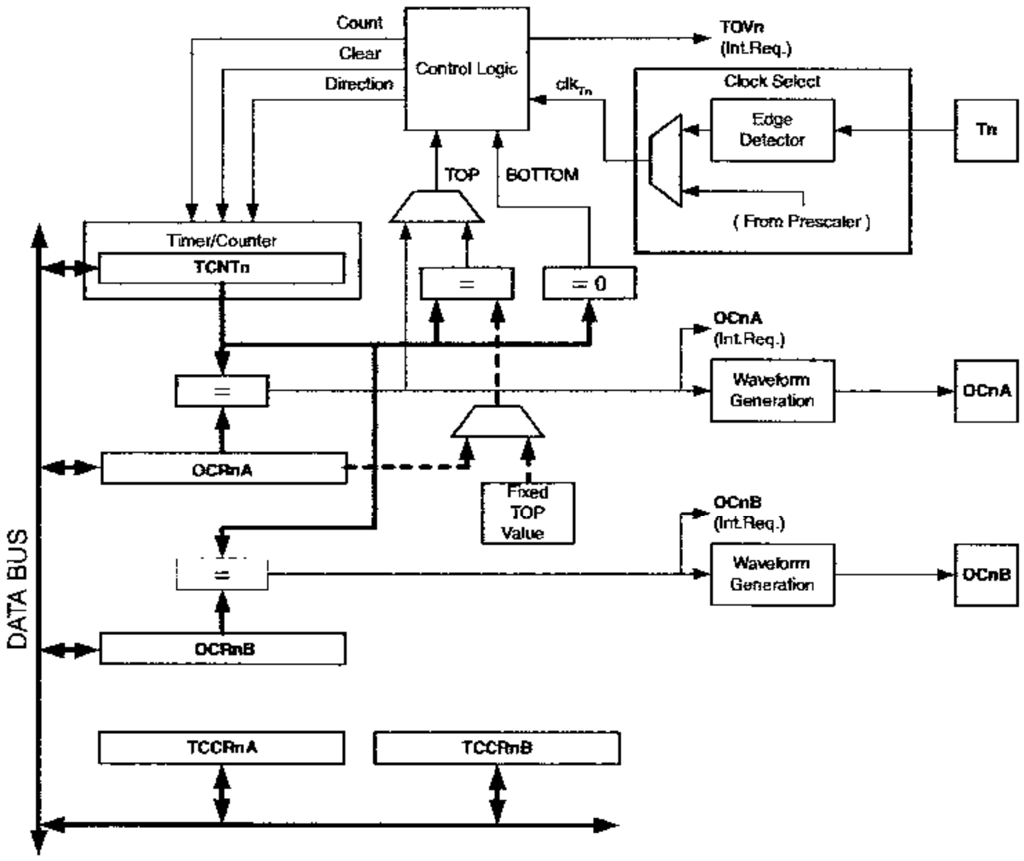


Figure 7.6: Timer 0 block diagram. (Figure used with permission Atmel, Inc.)

7.6.1 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 7.7. A specific mode of operation is selected using the Waveform Generation Mode bits located in Timer/Control Register A (TCCR0A) and Timer/Control Register B (TCCR0B).

7.6.1.1 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

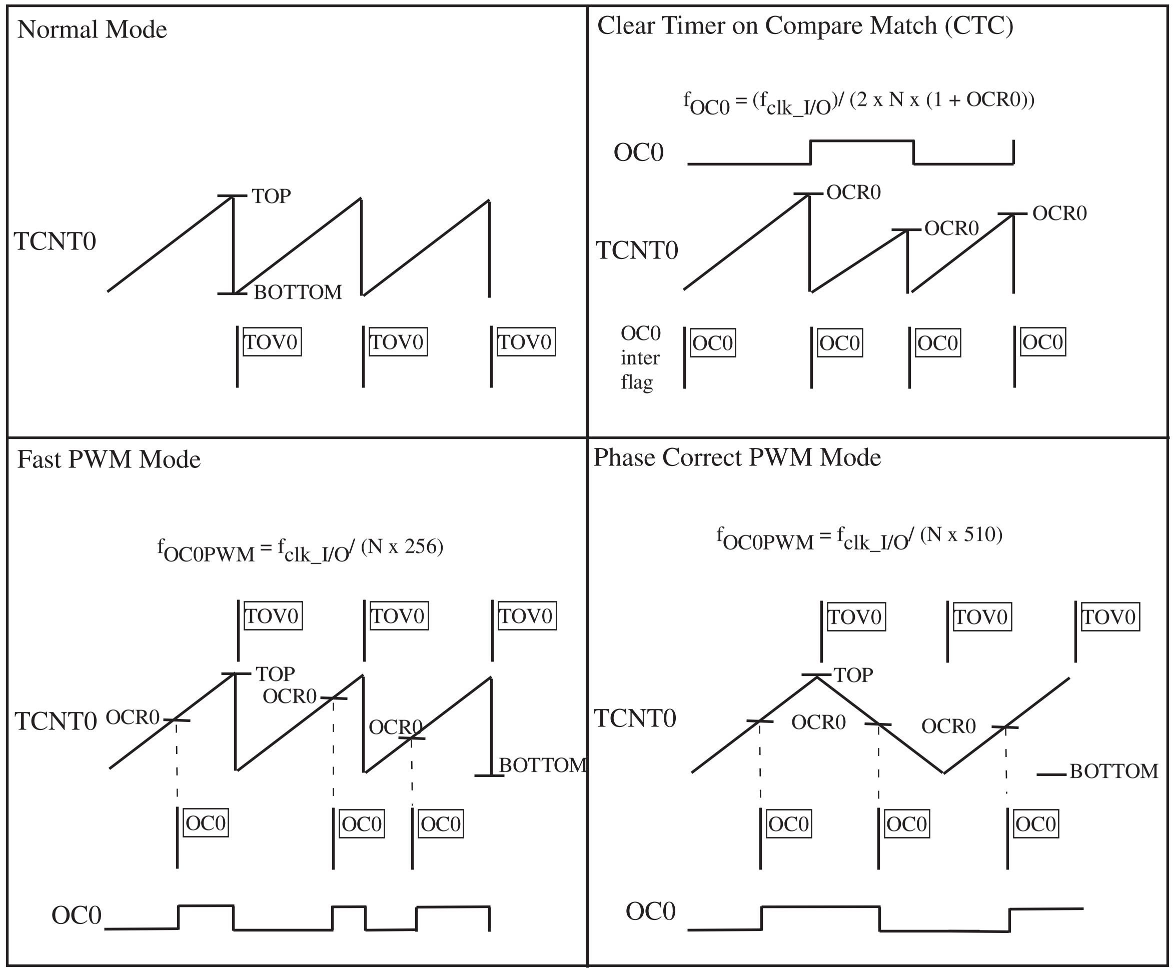


Figure 7.7: Timer 0 modes of operation.

(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.

7.6.1.2 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.

7.6.1.3 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.

7.6.1.4 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.

7.6.2 TIMER 0 REGISTERS

A summary of the Timer 0 registers are shown in Figure 7.8.

7.6.2.1 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 7.9.

Timer/Counter Control Register A (TCCR0A)

COM0A1	COM0A0	COM0B1	COM0B0	---	---	WGM01	WGM00
7							0

Timer/Counter Control Register B (TCCR0B)

FOC0A	FOC0B	---	---	WGM02	CS02	CS01	CS00
7							0

Timer/Counter Register (TCNT0)

7							0

Output Compare Register A (OCR0A)

7							0

Output Compare Register B (OCR0B)

7							0

Timer/Counter Interrupt Mask Register 0 (TIMSK0)

---	---	---	---	---	OCIE0B	OCIE0A	TOIE0
7							0

Timer/Counter Interrupt Flag REgister 0 (TIFR0)

---	---	---	---	---	OCF0B	OCF0A	TOV0
7							0

Figure 7.8: Timer 0 registers.

7.6.2.2 Timer/Counter Register(TCNT0)

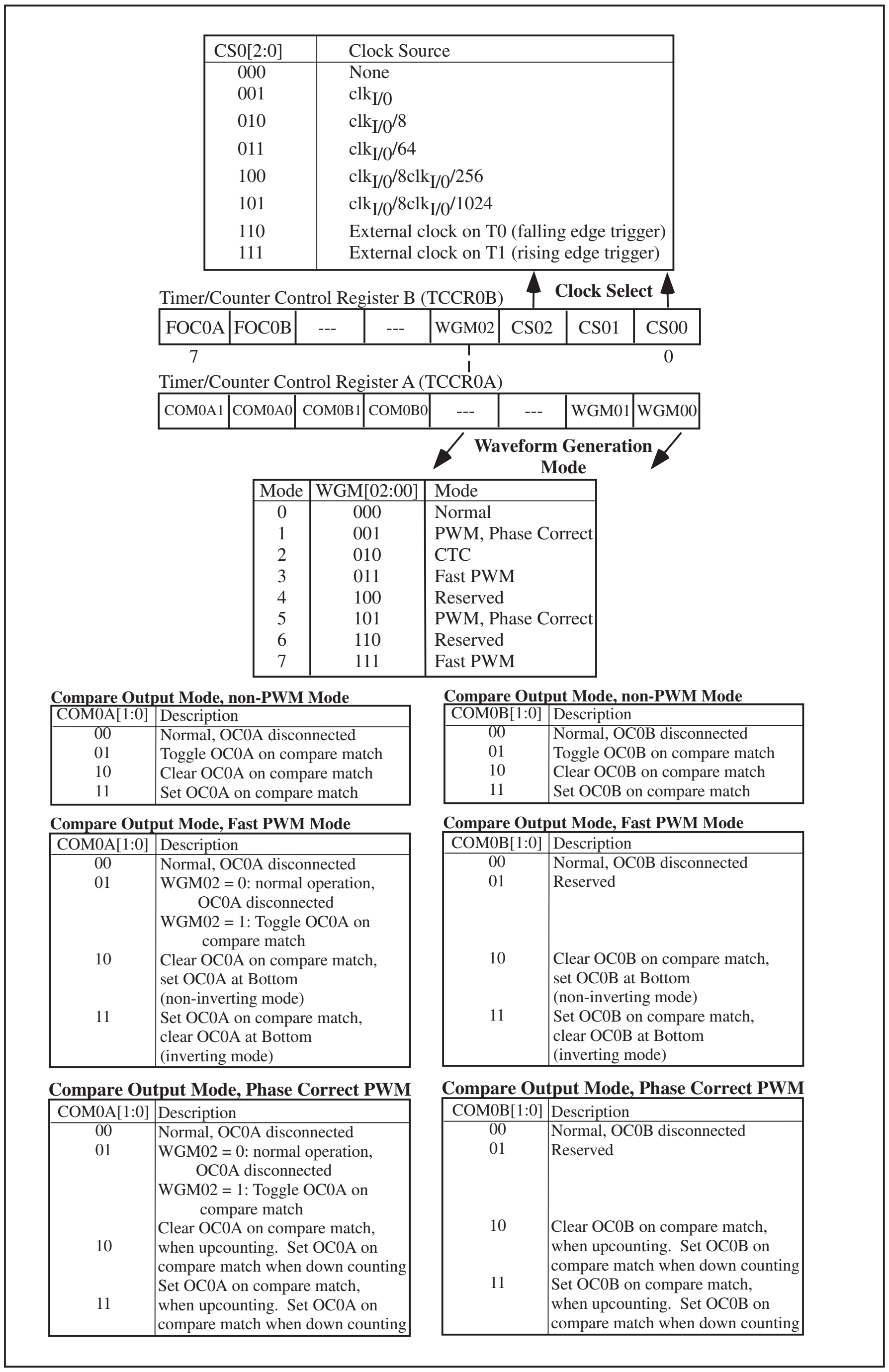
The TCNT0 is the 8-bit counter for Timer 0.

7.6.2.3 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.

7.6.2.4 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



CS0[2:0]	Clock Source
000	None
001	$clk_{I/0}$
010	$clk_{I/0}/8$
011	$clk_{I/0}/64$
100	$clk_{I/0}/8clk_{I/0}/256$
101	$clk_{I/0}/8clk_{I/0}/1024$
110	External clock on T0 (falling edge trigger)
111	External clock on T1 (rising edge trigger)

Timer/Counter Control Register B (TCCR0B)

7							0
FOC0A	FOC0B	---	---	WGM02	CS02	CS01	CS00

Timer/Counter Control Register A (TCCR0A)

COM0A1	COM0A0	COM0B1	COM0B0	---	---	WGM01	WGM00

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 7.9: Timer/Counter Control Registers A and B (TCCR0A and TCCR0B) bit settings.

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.

7.6.2.5 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.

7.7 TIMER 1

Timer 1 is a 16-bit timer/counter. It shares many of the same features of the Timer 0 channel. Due to limited space the shared information will not be repeated. Instead, we concentrate on the enhancements of Timer 1 which include an additional output compare channel and also the capability for input capture. The block diagram for Timer 1 is shown in Figure 7.10.

As discussed earlier in the chapter, the input capture feature is used to capture the characteristics of an input signal including period, frequency, duty cycle, or pulse length. This is accomplished by monitoring for a user-specified edge on the ICP1 microcontroller pin. When the desired edge occurs, the value of the Timer/Counter 1 (TCNT1) register is captured and stored in the Input Capture Register 1 (ICR1).

7.7.1 TIMER 1 REGISTERS

The complement of registers supporting Timer 1 are shown in Figure 7.11. Each register will be discussed in turn.

7.7.1.1 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 7.12.

7.7.1.2 Timer/Counter Register 1 (TCNT1H/TCNT1L)

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

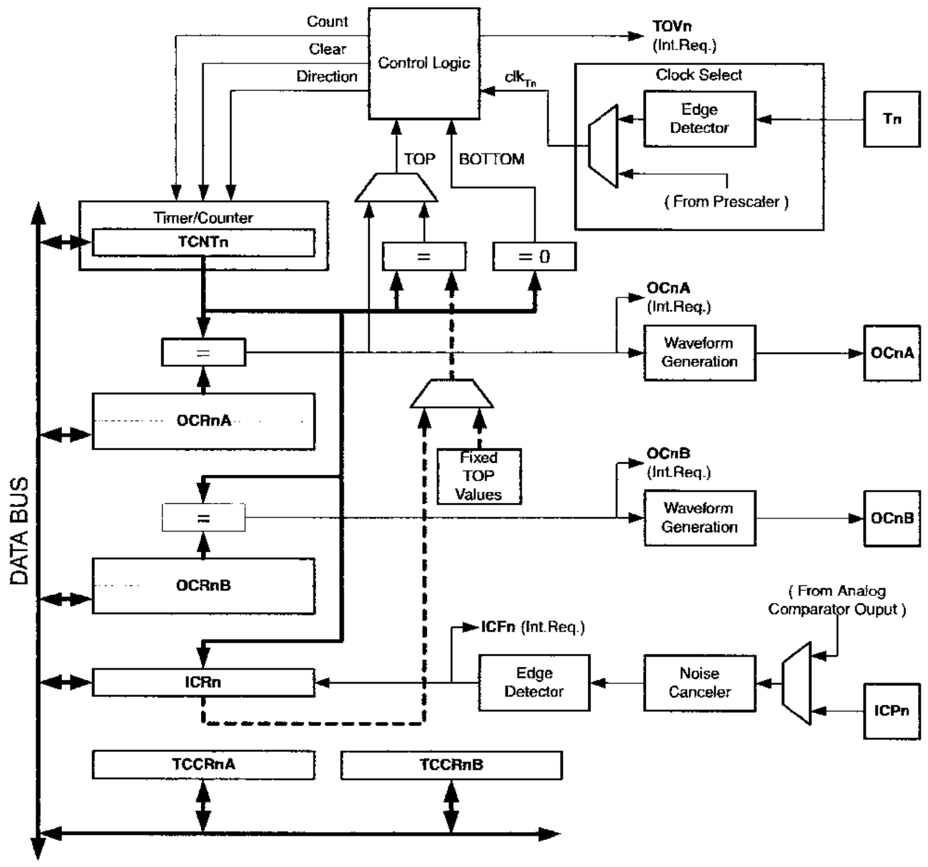


Figure 7.10: Timer 1 block diagram. (Figure used with Permission, Atmel,Inc.)

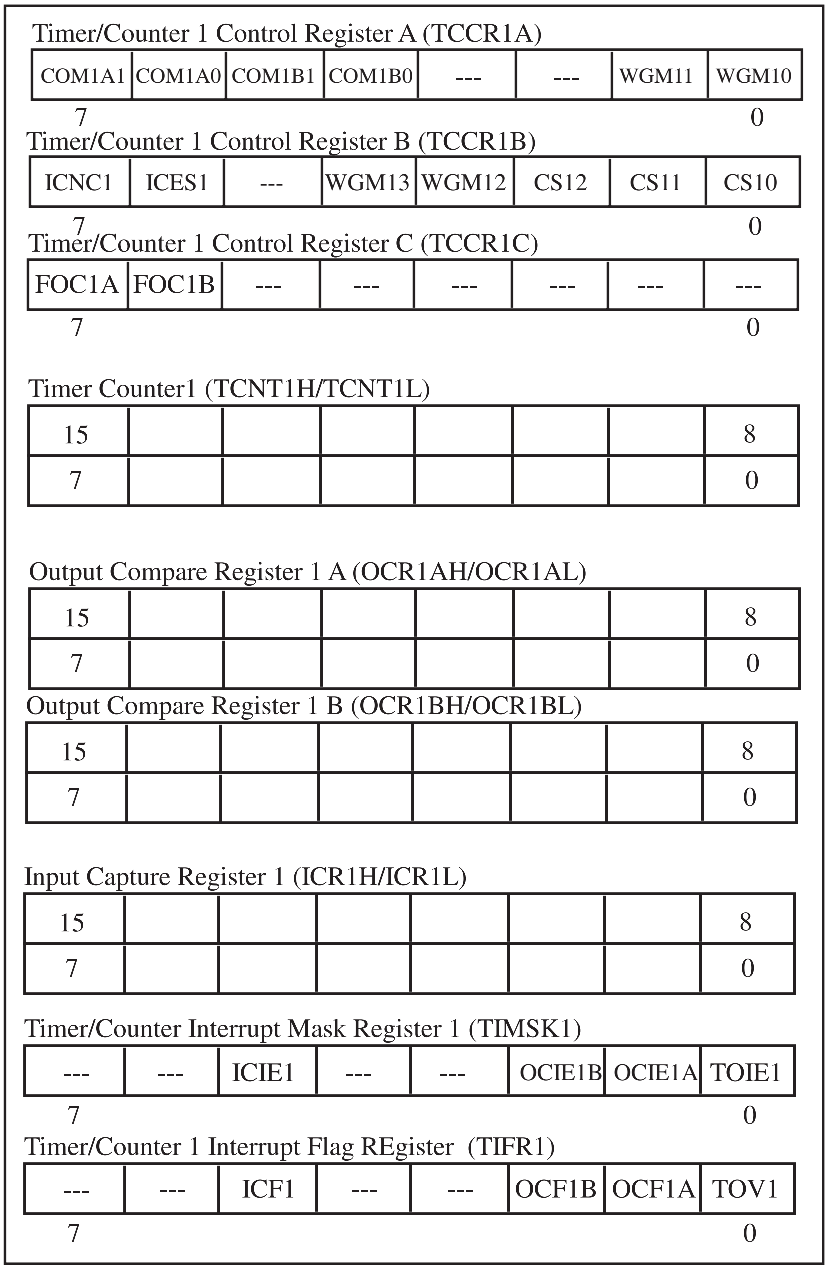
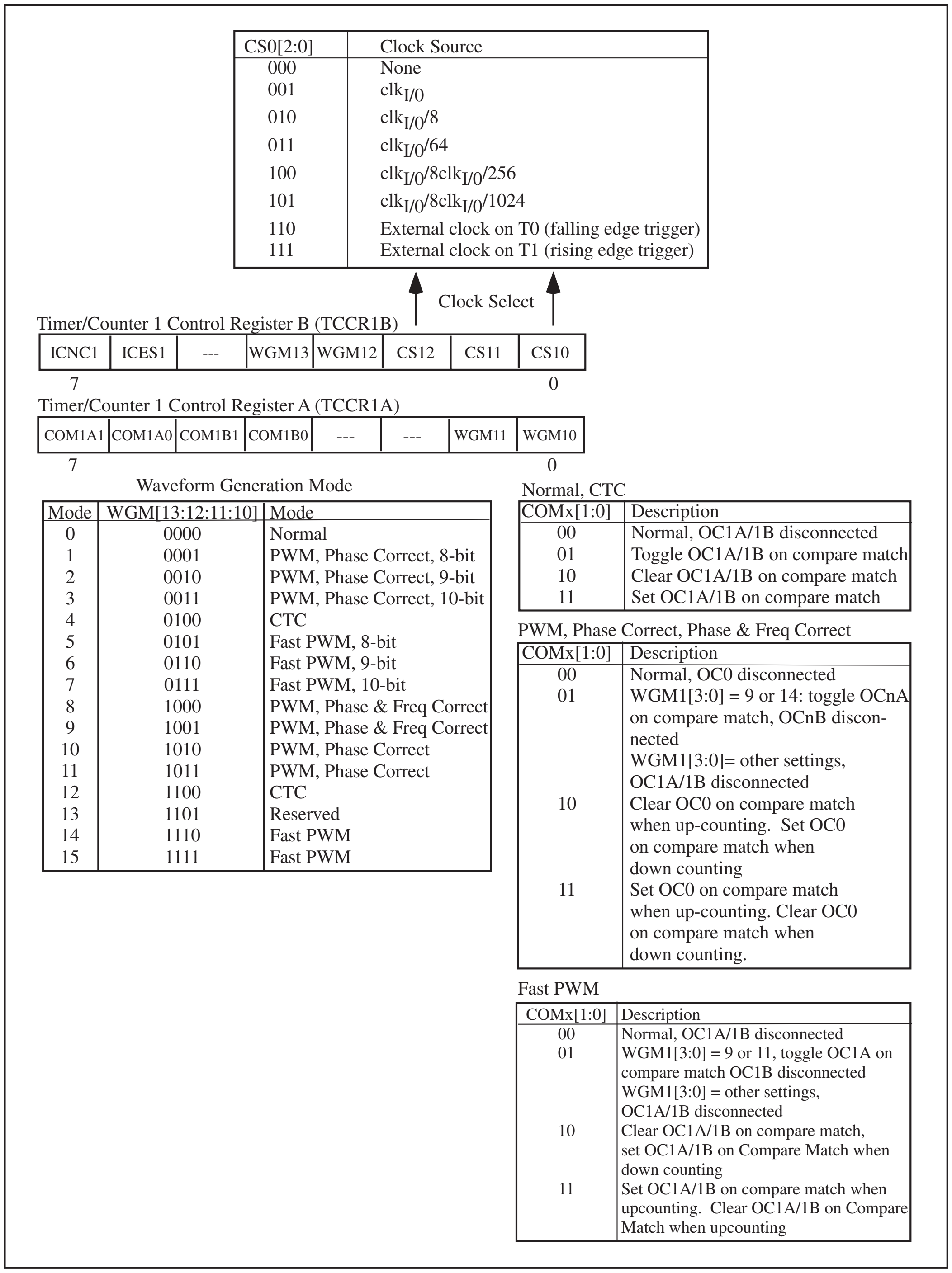

Timer/Counter 1 Control Register A (TCCR1A)
COM1A1 COM1A0 COM1B1 COM1B0 --- --- WGM11 WGM10
7 0
Timer/Counter 1 Control Register B (TCCR1B)
ICNC1 ICES1 --- WGM13 WGM12 CS12 CS11 CS10
7 0
Timer/Counter 1 Control Register C (TCCR1C)
FOC1A FOC1B --- --- --- --- --- ---
7 0
Timer Counter1 (TCNT1H/TCNT1L)
15 8
7 0
Output Compare Register 1 A (OCR1AH/OCR1AL)
15 8
7 0
Output Compare Register 1 B (OCR1BH/OCR1BL)
15 8
7 0
Input Capture Register 1 (ICR1H/ICR1L)
15 8
7 0
Timer/Counter Interrupt Mask Register 1 (TIMSK1)
--- --- ICIE1 --- --- OCIE1B OCIE1A TOIE1
7 0
Timer/Counter 1 Interrupt Flag REgister (TIFR1)
--- --- ICF1 --- --- OCF1B OCF1A TOV1
7 0

Figure 7.11: Timer 1 registers.



CS0[2:0]	Clock Source
000	None
001	$clk_{I/0}$
010	$clk_{I/0}/8$
011	$clk_{I/0}/64$
100	$clk_{I/0}/8clk_{I/0}/256$
101	$clk_{I/0}/8clk_{I/0}/1024$
110	External clock on T0 (falling edge trigger)
111	External clock on T1 (rising edge trigger)

Clock Select
Timer/Counter 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 7.12: TCCR1A and TCCR1B registers.

7.7.1.3 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.

7.7.1.4 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.

7.7.1.5 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.

7.7.1.6 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.

7.7.1.7 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.

7.8 TIMER 2

Timer 2 is another 8-bit timer channel similar to Timer 0. The Timer 2 channel block diagram is provided in Figure 7.13. Its registers are summarized in Figure 7.14.

7.8.0.8 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

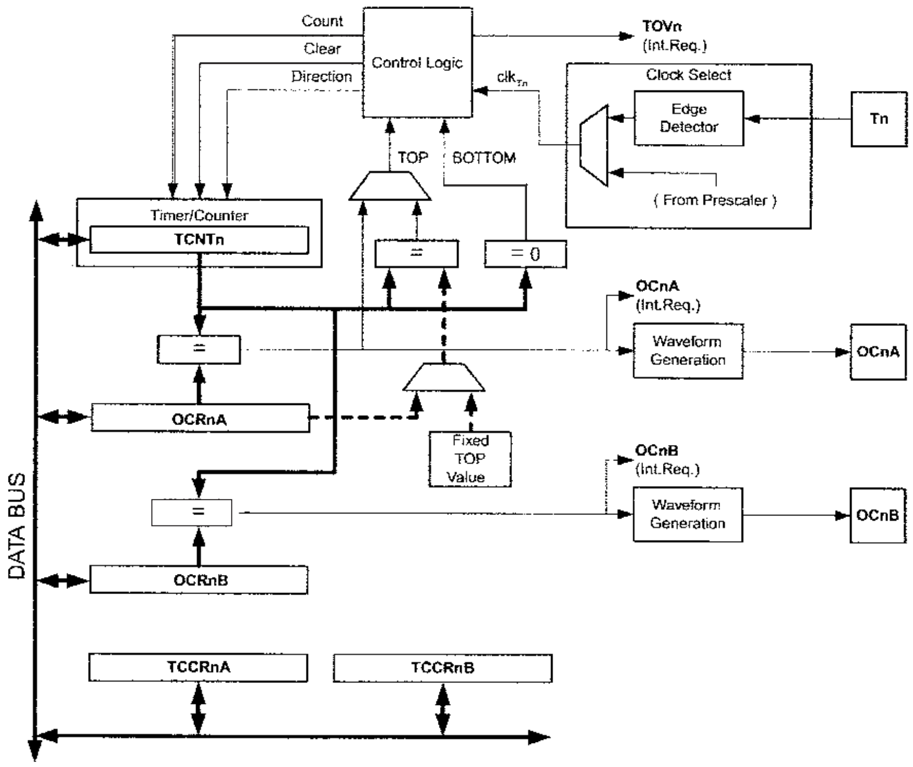


Figure 7.13: Timer 2 block diagram. (Figure used with Permission, Atmel,Inc.)

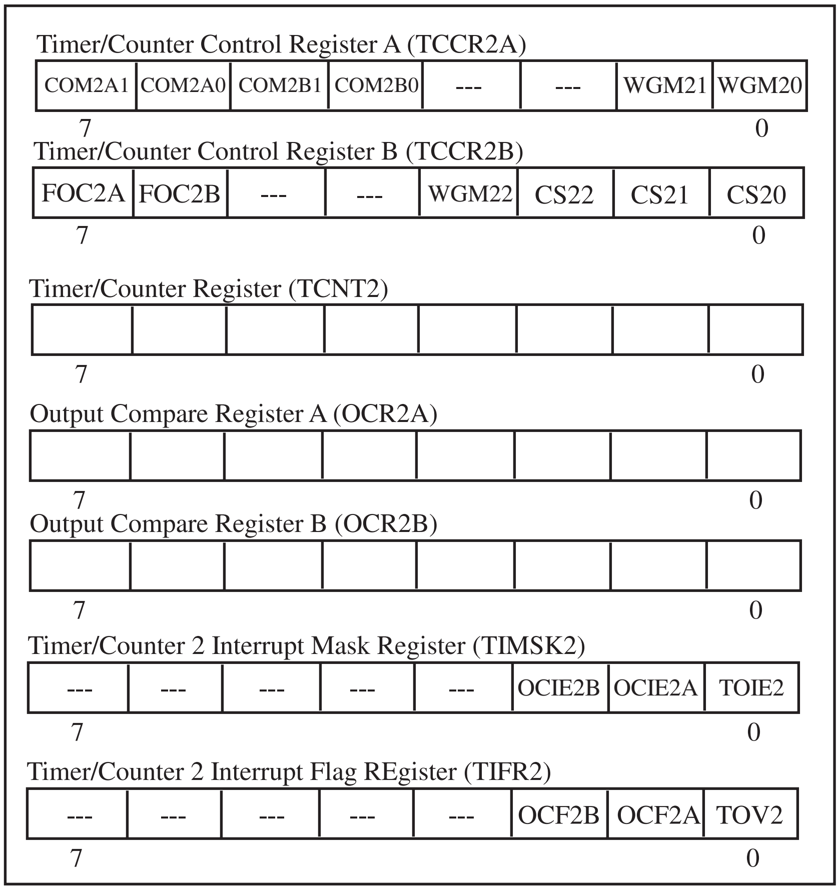

Timer/Counter Control Register A (TCCR2A)
COM2A1 COM2A0 COM2B1 COM2B0 --- --- WGM21 WGM20
7 0
Timer/Counter Control Register B (TCCR2B)
FOC2A FOC2B --- --- WGM22 CS22 CS21 CS20
7 0
Timer/Counter Register (TCNT2)
7 0
Output Compare Register A (OCR2A)
7 0
Output Compare Register B (OCR2B)
7 0
Timer/Counter 2 Interrupt Mask Register (TIMSK2)
--- --- --- --- --- OCIE2B OCIE2A TOIE2
7 0
Timer/Counter 2 Interrupt Flag REgister (TIFR2)
--- --- --- --- --- OCF2B OCF2A TOV2
7 0

Figure 7.14: Timer 2 registers.

- 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 7.15.

7.8.0.9 Timer/Counter Register(TCNT2)

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

7.8.0.10 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.

7.8.0.11 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.

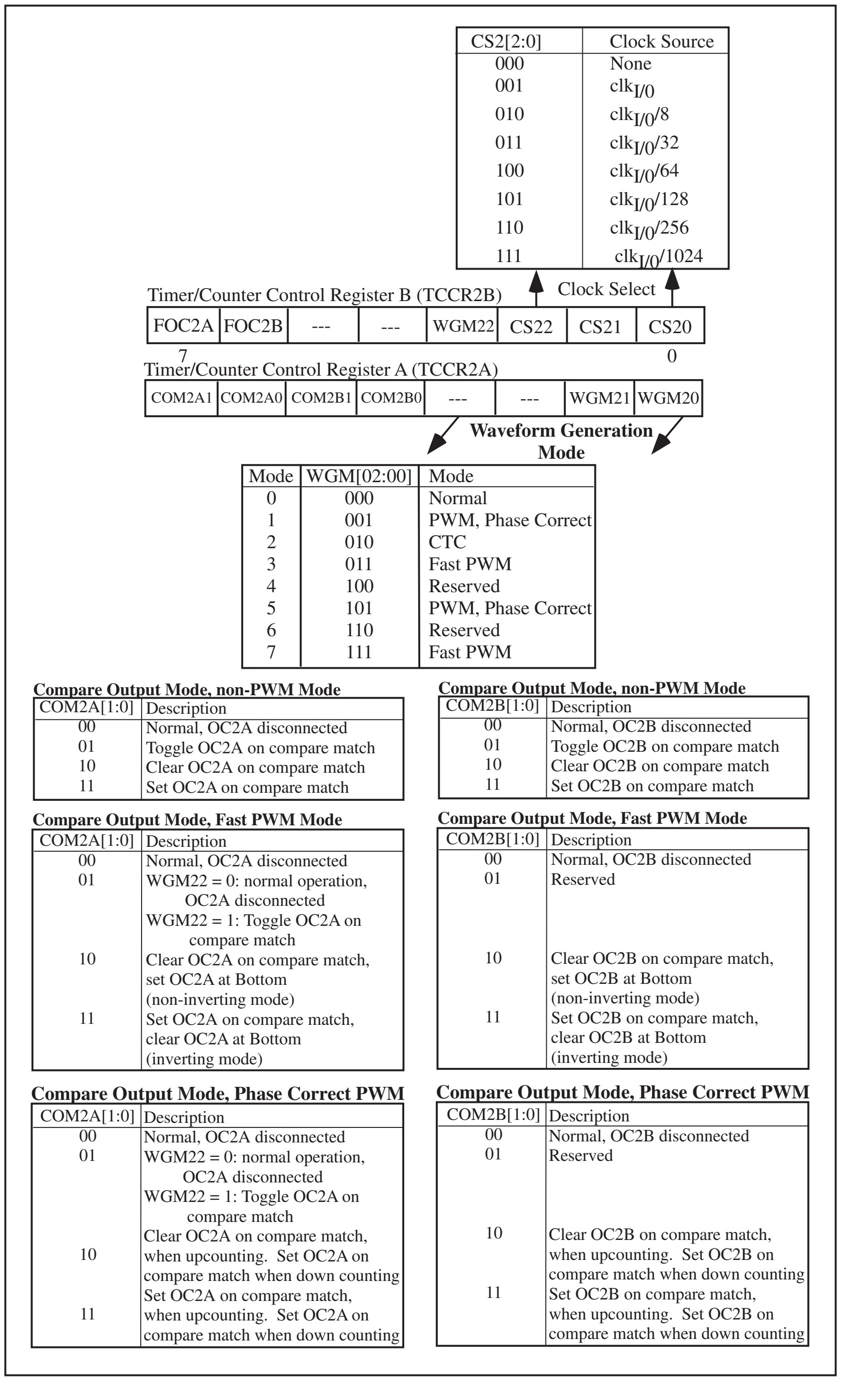
7.8.0.12 Timer/Counter Interrupt Flag Register 2 (TIFR2)

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.

7.9 PROGRAMMING THE ARDUINO DUEMILANOVE USING THE BUILT-IN ARDUINO DEVELOPMENT ENVIRONMENT TIMING FEATURES

The Arduino Development Environment has several built-in timing features. These include:

- **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) percent.



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

Timer/Counter Control Register B (TCCR2B)

FOC2A	FOC2B	---	---	WGM22	CS22	CS21	CS20

Timer/Counter Control Register A (TCCR2A)

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

Mode	WGM[02:00]	Mode
0	000	Normal
1	001	PWM, Phase Correct
2	010	CTC
3	011	Fast PWM
4	100	Reserved
5	101	PWM, Phase Correct
6	110	Reserved
7	111	Fast PWM

Compare Output Mode, non-PWM Mode

COM2A[1:0]	Description
00	Normal, OC2A disconnected
01	Toggle OC2A on compare match
10	Clear OC2A on compare match
11	Set OC2A on compare match

Compare Output Mode, non-PWM Mode

COM2B[1:0]	Description
00	Normal, OC2B disconnected
01	Toggle OC2B on compare match
10	Clear OC2B on compare match
11	Set OC2B on compare match

Compare Output Mode, Fast PWM Mode

COM2A[1:0]	Description
00	Normal, OC2A disconnected

01	WGM22 = 0: normal operation, OC2A disconnected
	WGM22 = 1: Toggle OC2A on compare match

10	Clear OC2A on compare match, set OC2A at Bottom (non-inverting mode)
11	Set OC2A on compare match, clear OC2A at Bottom (inverting mode)

Compare Output Mode, Fast PWM Mode

COM2B[1:0]	Description
00	Normal, OC2B disconnected
01	Reserved
10	Clear OC2B on compare match, set OC2B at Bottom (non-inverting mode)
11	Set OC2B on compare match, clear OC2B at Bottom (inverting mode)

Compare Output Mode, Phase Correct PWM

COM2A[1:0]	Description
00	Normal, OC2A disconnected

01	WGM22 = 0: normal operation, OC2A disconnected
	WGM22 = 1: Toggle OC2A on compare match

10	Clear OC2A on compare match, when upcounting. Set OC2A on compare match when down counting
11	Set OC2A on compare match, when upcounting. Set OC2A on compare match when down counting

Compare Output Mode, Phase Correct PWM

COM2B[1:0]	Description
00	Normal, OC2B disconnected
01	Reserved
10	Clear OC2B on compare match, when upcounting. Set OC2B on compare match when down counting
11	Set OC2B on compare match, when upcounting. Set OC2B on compare match when down counting

Figure 7.15: Timer/Counter Control Register A and B (TCCR2A and B) bit settings.

We have already used the analogWrite function in an earlier chapter to control the motor speed of the Blinky 602A robot.

7.10 PROGRAMMING THE TIMER SYSTEM IN C

In this section, we provide 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.

7.10.1 PRECISION DELAY IN C

In this example, we program the ATmega328 to provide a delay of some number of 6.55 ms interrupts. The Timer 0 overflow is configured to occur every 6.55 ms. The overflow flag is used as a "clock tick" to generate a precision delay. To create the delay the microcontroller is placed in a while loop waiting for the prescribed number of Timer 0 overflows to occur.

```
//Function prototypes
void delay(unsigned int number_of_6_55ms_interrupts);
void init_timer0_ovf_interrupt(void);
void timer0_interrupt_isr(void);


                                      //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:17

//door profile data


//*************************************************************************
//int_timer0_ovf_interrupt(): The Timer0 overflow interrupt is being
//employed as a time base for a master timer for this project.
//The ceramic resonator operating at 10 MHz is divided by 256.
//The 8-bit Timer0 register (TCNT0) overflows every 256 counts
//or every 6.55 ms.
//*************************************************************************

void init_timer0_ovf_interrupt(void)
{
TCCR0B = 0x04; //divide timer0 timebase by 256, overfl. occurs every 6.55ms
TIMSK0 = 0x01; //enable timer0 overflow interrupt
asm("SEI");    //enable global interrupt
}

//*************************************************************************
//*************************************************************************
//timer0_interrupt_isr:
//Note: Timer overflow 0 is cleared by hardware when executing the
//corresponding interrupt handling vector.
//*************************************************************************


void timer0_interrupt_isr(void)
{
input_delay++;     //input delay processing
}

//*************************************************************************
//*************************************************************************
//delay(unsigned int num_of_6_55ms_interrupts): this generic delay function
//provides the specified delay as the number of 6.55 ms "clock ticks" from
//the Timer0 interrupt.
//Note: this function is only valid when using a 10 MHz crystal or ceramic
//      resonator
//*************************************************************************

void delay(unsigned int number_of_6_55ms_interrupts)
{
TCNT0 = 0x00;                  //reset timer0
input_delay = 0;
while(input_delay <= number_of_6_55ms_interrupts)
  {
  ;
  }
}

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

7.10.2 PULSE WIDTH MODULATION IN C

The function provided below is used to configure output compare channel B to generate a pulse width modulated signal. An analog voltage provided to ADC Channel 3 is used to set the desired duty cycle from 50 to 100 percent. Note how the PWM ramps up from 0 to the desired speed.

```
//Function Prototypes
void PWM(unsigned int PWM_incr)
{
unsigned int  Open_Speed_int;
float         Open_Speed_float;
int           gate_position_int;


PWM_duty_cycle = 0;
InitADC();                               //Initialize ADC

                                         //Read "Open Speed" volt
Open_Speed_int = ReadADC(0x03);          //setting PA3

                                          //Open Speed Setting unsigned int

                                          //Convert to max duty cycle
 //setting

                                          //0 VDC = 50% = 127,
                                          //5 VDC = 100% =255
Open_Speed_float = ((float)(Open_Speed_int)/(float)(0x0400));

                                          //Convert volt to PWM constant
                                          //127-255
Open_Speed_int = (unsigned int)((Open_Speed_float * 127) + 128.0);
                                          //Configure PWM clock
TCCR1A = 0xA1;                            //freq = resonator/510 = 10MHz/510

                                          //freq = 19.607 kHz
TCCR1B = 0x01;                            //no clock source division

PWM_duty_cycle = 0;                       //Initiate PWM duty cycle
                                          //variables
```

```
OCR1BH = 0x00;
OCR1BL = (unsigned char)(PWM_duty_cycle);//Set PWM duty cycle CH B to 0%
                                         //Ramp up to Open Speed in 1.6s
OCR1BL = (unsigned char)(PWM_duty_cycle);//Set PWM duty cycle CH B

while (PWM_duty_cycle < Open_Speed_int)
  {
  if(PWM_duty_cycle < Open_Speed_int)    //Increment duty cycle
    PWM_duty_cycle=PWM_duty_cycle + PWM_open_incr;
                                         //Set PWM duty cycle CH B
  OCR1BL = (unsigned char)(PWM_duty_cycle);
  }

//Gate continues to open at specified upper speed (PA3)
:
:
:

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

7.10.3 INPUT CAPTURE MODE IN C

This example was 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 is being used to monitor the heart rate (typically 50-120 beats per minute) of a patient. The microcontroller is set to operate at an internal clock frequency of 1 MHz. Timer/Counter channel 1 is used in this example.

```
//*********************************************************************
//initialize_ICP_interrupt: Initialize Timer/Counter 1 for input capture
//*********************************************************************

void initialize_ICP_interrupt(void)
{
TIMSK=0x20;                         //Allows input capture interrupts
SFIOR=0x04;                         //Internal pull-ups disabled
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)
{
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
TIFR=0x20;                              //Clear the input capture flag
asm("RETI");                            //Resets the I flag to allow
                                        //global interrupts
}

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

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 = 1.024 ms
    {                               //Divide by 977 to get seconds/pulse
     HR=60/(time_pulses/977);       //(secs/min)/(secs/beat) =bpm
     }
  else
     {
     HR=0;
     }
  }
else
  {
  HR=0;
  }
}

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

7.11 SERVO MOTOR CONTROL WITH THE PWM SYSTEM IN C

A servo motor provides an angular displacement from 0 to 180 degrees. Most servo motors provide the angular displacement relative to the pulse length of repetitive pulses sent to the motor as shown in Figure 7.16. A 1 ms pulse provides an angular displacement of 0 degrees while a 2 ms pulse provides a displacement of 180 degrees. Pulse lengths in between these two extremes provide angular displacements between 0 and 180 degrees. Usually, a 20 to 30 ms low signal is provided between the active pulses.

A test and interface circuit for a servo motor is provided in Figure 7.16. 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.

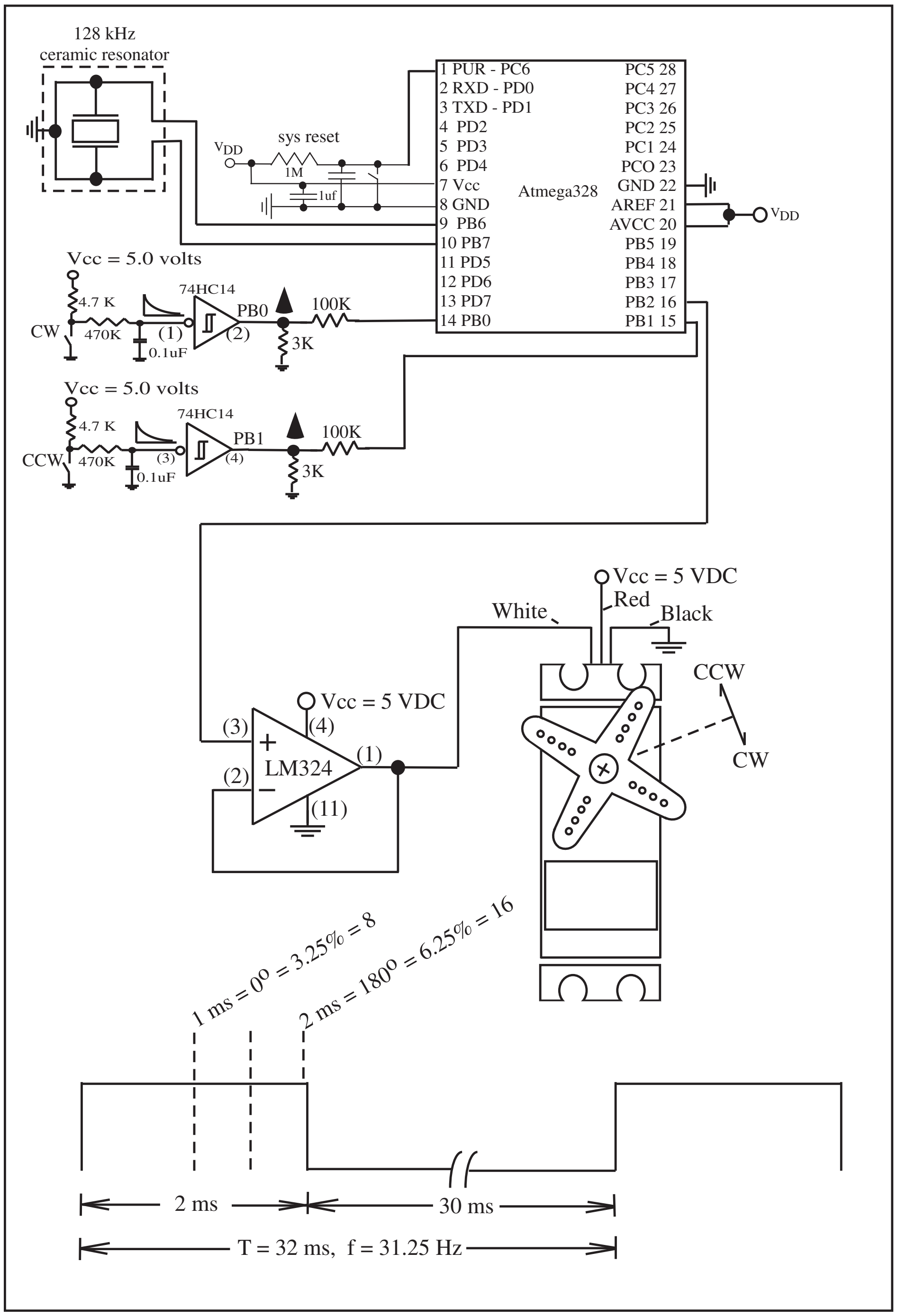


Figure 7.16: Test and interface circuit for a servo motor.

```
//*************************************************************************
//target controller: ATMEL ATmega328
//
//ATMEL 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

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

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

//function prototypes*****************************************************
void initialize_ports(void);           //initializes ports
void read_new_input(void);             //used to read input change on PORTB
void init_timer0_ovf_interrupt(void);  //used to initialize 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();
 //return LED configuration 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% =
                                       //1 ms = 0 degrees = 8 counts to

 //6.2% = 2 ms = 180 degrees = 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 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=0xfc;                            //PORTB[7-2] output, PORTB[1:0] input
PORTB=0x00;                           //disable PORTB pull-up resistors

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

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

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

//read_new_input: functions polls PORTB for a change in status. If status
//change has occurred, appropriate function for status change is called
//Pin 1 PB0 to active high RC  debounced switch - CW
//Pin 2 PB1 to active high RC debounced switch  - CCW
//*************************************************************************

void read_new_input(void)
{
new_PORTB = (PINB);
if(new_PORTB != old_PORTB){
  switch(new_PORTB){                   //process change in PORTB input
    case 0x01:                         //CW
      while(PINB == 0x01)
        {
        PWM_duty_cycle = PWM_duty_cycle + 1;
        if(PWM_duty_cycle > 16) PWM_duty_cycle = 16;
        OCR1BH = 0x00;
        OCR1BL = (unsigned char)(PWM_duty_cycle);
        }
```

```
     break;

     case 0x02:                        //CCW
       while(PINB == 0x02)
         {
         PWM_duty_cycle = PWM_duty_cycle - 1;
         if(PWM_duty_cycle < 8) PWM_duty_cycle = 8;
         OCR1BH = 0x00;
         OCR1BL = (unsigned char)(PWM_duty_cycle);
         }
      break;


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

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

7.12 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 timer channels. We then investigated the built-in timing features of the Arduino Development Environment. We concluded with an example employing a servo motor controlled by the ATmega328 PWM system.

7.13 REFERENCES

- Kenneth Short, *Embedded Microprocessor Systems Design: An Introduction Using the INTEL 80C188EB*, Prentice Hall, Upper Saddle River, 1998.

- Frederick Driscoll, Robert Coughlin, and Robert Villanucci, *Data Acquisition and Process Control with the M68HC11 Microcontroller,* Second Edition, Prentice Hall, Upper Saddle River, 2000.

- Todd Morton, *Embedded Microcontrollers,* Prentice Hall, Upper Saddle River, Prentice Hall, 2001.
- *Atmel 8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash, ATmega328, ATmega328L,* data sheet: 2466L-AVR-06/05, Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131.
- Barrett S, Pack D (2006) Microcontrollers Fundamentals for Engineers and Scientists. Morgan and Claypool Publishers. DOI: 10.2200/S00025ED1V01Y200605DCS001
- Barrett S and Pack D (2008) Atmel AVR Microcontroller Primer Programming and Interfacing. Morgan and Claypool Publishers. DOI: 10.2200/S00100ED1V01Y200712DCS015
- Barrett S (2010) Embedded Systems Design with the Atmel AVR Microcontroller. Morgan and Claypool Publishers. DOI: 10.2200/S00225ED1V01Y200910DCS025

7.14 CHAPTER PROBLEMS

1. Given an 8 bit free running counter and the system clock rate of 24 MHz, find the time required for the counter to count from zero to its maximum value.
2. If we desire to generate periodic signals with periods ranging from 125 nanoseconds to 500 microseconds, what is the minimum frequency of the system clock?
3. Describe how you can compute the period of an incoming signal with varying duty cycles.
4. Describe how one can generate an aperiodic pulse with a pulse width of 2 minutes?
5. Program the output compare system of the ATmega328 to generate a 1 kHz signal with a 10 percent duty cycle.
6. 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 millisecond to turn on the sprinklers in zone one.
   (b) After 15 minutes, your microcontroller must send the PWM signal with 15% duty cycle for 10 millisecond to turn off the sprinklers in zone one.
   (c) After 15 minutes, your microcontroller must send the PWM signal with 20% duty cycle for 10 millisecond to turn on the sprinklers in zone two.

(d) After 15 minutes, your microcontroller must send the PWM signal with 25% duty cycle for 10 millisecond to turn off the sprinklers in zone two.

7. Modify the servo motor example to include a potentiometer connected to PORTA[0]. The servo will deflect 0 degrees for 0 VDC applied to PORTA[0] and 180 degrees for 5 VDC.

8. For the automated cooling fan example, what would be the effect of changing the PWM frequency applied to the fan?

9. Modify the code of the automated cooling fan example to also display the set threshold temperature.

CHAPTER 8
Atmel AVR Operating Parameters and Interfacing

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

- Describe the voltage and current parameters for the Arduino Duemilanove and the Atmel AVR HC CMOS type microcontroller.
- Specify a battery system to power an Arduino Duemilanove and the Atmel AVR based system.
- Apply the voltage and current parameters toward properly interfacing input and output devices to the Arduino Duemilanove and the Atmel AVR microcontroller.
- Interface a wide variety of input and output devices to the Arduino Duemilanove and the Atmel AVR microcontroller.
- Describe the special concerns that must be followed when the Arduino Duemilanove and the Atmel AVR microcontroller is used to interface to a high power DC or AC device.
- Discuss the requirement for an optical based interface.
- Describe how to control the speed and direction of a DC motor.
- Describe how to control several types of AC loads.

8.1 OVERVIEW

The textbook for Morgan & Claypool Publishers (M&C) titled, "Microcontrollers Fundamentals for Engineers and Scientists," contains a chapter entitled "Operating Parameters and Interfacing." With M&C permission, we repeated portions of the chapter here for your convenience. However, we have customized the information provided to the Arduino Duemilanove and the Atmel AVR line of microcontrollers and have also expanded the coverage of the chapter to include interface techniques for a number of additional input and output devices.

In this chapter, we introduce you to the extremely important concepts of the operating envelope for a microcontroller. We begin by reviewing the voltage and current electrical parameters for the HC CMOS based Atmel AVR line of microcontrollers. We then show how to apply this information to properly interface input and output devices to the Arduino Duemilanove and the ATmega328 microcontroller. We then discuss the special considerations for controlling a high power DC or AC load such as a motor and introduce the concept of an optical interface. Throughout the chapter, we provide a number of detailed examples.

The importance of this chapter can not be emphasized enough. Any time an input or an output device is connected to a microcontroller, the interface between the device and the microcontroller must be carefully analyzed and designed. This will ensure the microcontroller will continue to operate within specified parameters. Should the microcontroller be operated outside its operational envelope, erratic, unpredictable, and unreliable system may result.

8.2 OPERATING PARAMETERS

Any time a device is connected to a microcontroller, careful interface analysis must be performed. Most microcontrollers are members of the "HC," or high-speed CMOS, family of chips. As long as all components in a system are also of the "HC" family, as is the case for the Arduino Duemilanove and the Atmel AVR line of microcontrollers, electrical interface issues are minimal. If the microcontroller is connected to some component not in the "HC" family, electrical interface analysis must be completed. Manufacturers readily provide the electrical characteristic data necessary to complete this analysis in their support documentation.

To perform the interface analysis, there are eight different electrical specifications required for electrical interface analysis. The electrical parameters are:

- V_{OH}: the lowest guaranteed output voltage for a logic high,
- V_{OL}: the highest guaranteed output voltage for a logic low,
- I_{OH}: the output current for a V_{OH} logic high,
- I_{OL}: the output current for a V_{OL} logic low,
- V_{IH}: the lowest input voltage guaranteed to be recognized as a logic high,
- V_{IL}: the highest input voltage guaranteed to be recognized as a logic low,
- I_{IH}: the input current for a V_{IH} logic high, and
- I_{IL}: the input current for a V_{IL} logic low.

These electrical characteristics are required for both the microcontroller and the external components. Typical values for a microcontroller in the HC CMOS family assuming V_{DD} = 5.0 volts and V_{SS} = 0 volts are provided below. The minus sign on several of the currents indicates a current flow out of the device. A positive current indicates current flow into the device.

- V_{OH} = 4.2 volts,
- V_{OL} = 0.4 volts,

- I_{OH} = -0.8 milliamps,
- I_{OL} = 1.6 milliamps,
- V_{IH} = 3.5 volts,
- V_{IL} = 1.0 volt,
- I_{IH} = 10 microamps, and
- I_{IL} = -10 microamps.

It is important to realize that these are static values taken under very specific operating conditions. If external circuitry is connected such that the microcontroller acts as a current source (current leaving the microcontroller) or current sink (current entering the microcontroller), the voltage parameters listed above will also be affected.

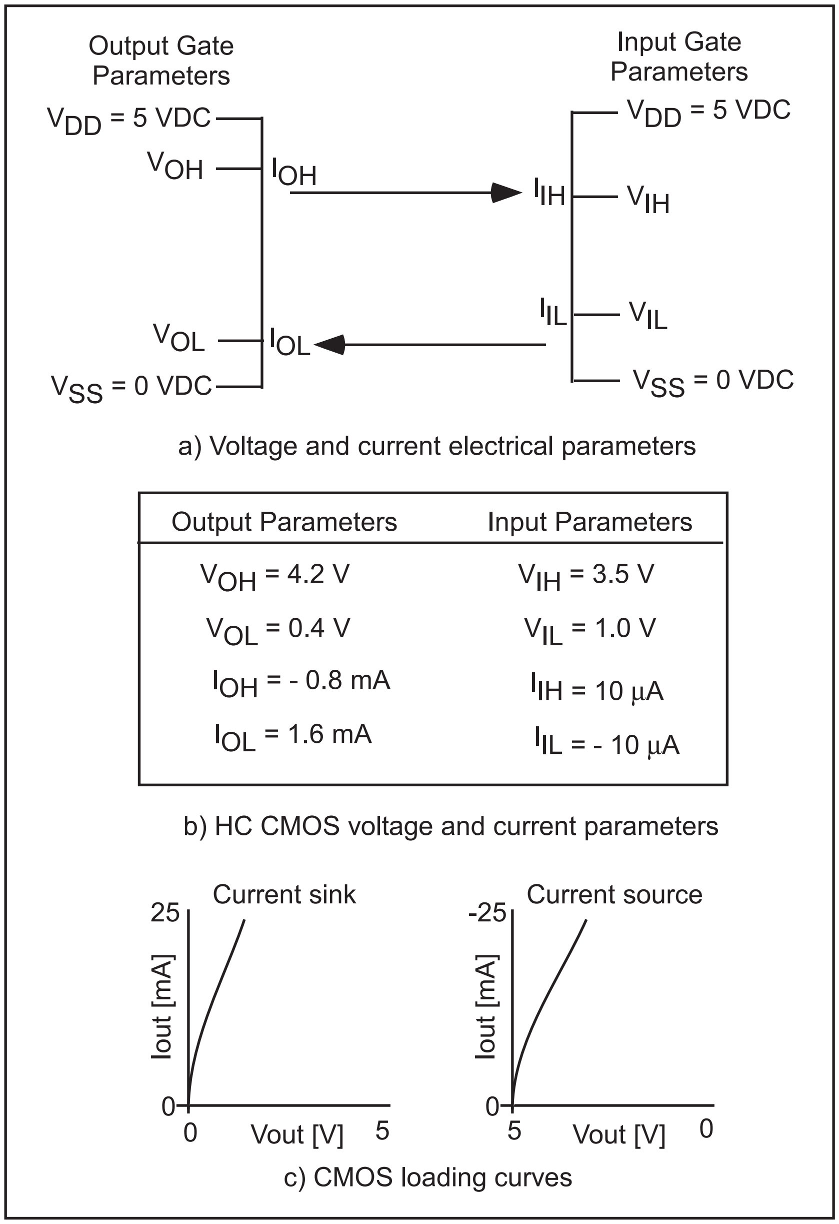
In the current source case, an output voltage V_{OH} is provided at the output pin of the microcontroller when the load connected to this pin draws a current of I_{OH}. If a load draws more current from the output pin than the I_{OH} specification, the value of V_{OH} is reduced. If the load current becomes too high, the value of V_{OH} falls below the value of V_{IH} for the subsequent logic circuit stage and not be recognized as an acceptable logic high signal. When this situation occurs, erratic and unpredictable circuit behavior results.

In the sink case, an output voltage V_{OL} is provided at the output pin of the microcontroller when the load connected to this pin delivers a current of I_{OL} to this logic pin. If a load delivers more current to the output pin of the microcontroller than the I_{OL} specification, the value of V_{OL} increases. If the load current becomes too high, the value of V_{OL} rises above the value of V_{IL} for the subsequent logic circuit stage and not be recognized as an acceptable logic low signal. As before, when this situation occurs, erratic and unpredictable circuit behavior results.

For convenience this information is illustrated in Figure 8.1. In (a), we provided an illustration of the direction of current flow from the HC device and also a comparison of voltage levels. As a reminder current flowing out of a device is considered a negative current (source case) while current flowing into the device is considered positive current(sink case). The magnitude of the voltage and current for HC CMOS devices are shown in (b). As more current is sinked or sourced from a microcontroller pin, the voltage will be pulled up or pulled down, respectively, as shown in (c). If input and output devices are improperly interfaced to the microcontroller, these loading conditions may become excessive, and voltages will not be properly interpreted as the correct logic levels.

You must also ensure that total current limits for an entire microcontroller port and overall bulk port specifications are complied with. For planning purposes the sum of current sourced or sinked from a port should not exceed 100 mA. Furthermore, the sum of currents for all ports should not exceed 200 mA. As before, if these guidelines are not followed, erratic microcontroller behavior may result.

The procedures presented in the following sections, when followed carefully, will ensure the microcontroller will operate within its designed envelope. The remainder of the chapter is divided into



Output Parameters	Input Parameters
$V_{OH} = 4.2$ V	$V_{IH} = 3.5$ V
$V_{OL} = 0.4$ V	$V_{IL} = 1.0$ V
$I_{OH} = -0.8$ mA	$I_{IH} = 10\ \mu$A
$I_{OL} = 1.6$ mA	$I_{IL} = -10\ \mu$A

Figure 8.1: Electrical voltage and current parameters.

input device interface analysis followed by output device interface analysis. Since many embedded systems operate from a DC battery source, we begin by examining several basic battery supply circuits.

8.3 BATTERY OPERATION

Many embedded systems are remote, portable systems operating from a battery supply. To properly design a battery source for an embedded system, the operating characteristics of the embedded system must be matched to the characteristics of the battery supply.

8.3.1 EMBEDDED SYSTEM VOLTAGE AND CURRENT DRAIN SPECIFICATIONS

An embedded system has a required supply voltage and an overall current requirement. For the purposes of illustration, we will assume our microcontroller based embedded system operates from 5 VDC. The overall current requirements of the system is determined by the worst case current requirements when all embedded system components are operational.

8.3.2 BATTERY CHARACTERISTICS

To properly match a battery to an embedded system, the battery voltage and capacity must be specified. Battery capacity is typically specified as a mAH rating. For example, a typical 9 VDC non-rechargeable alkaline battery has a capacity of 550 mAH. If the embedded system has a maximum operating current of 50 mA, it will operate for approximately eleven hours before battery replacement is required.

A battery is typically used with a voltage regulator to maintain the voltage at a prescribed level. Figure 8.2 provides sample circuits to provide a +5 VDC and a ±5 VDC portable battery source. Additional information on battery capacity and characteristics may be found in Barrett and Pack [Prentice-Hall, 2005].

8.4 INPUT DEVICES

In this section, we discuss how to properly interface input devices to a microcontroller. We will start with the most basic input component, a simple on/off switch.

8.4.1 SWITCHES

Switches come in a variety of types. As a system designer it is up to you to choose the appropriate switch for a specific application. Switch varieties commonly used in microcontroller applications are illustrated in Figure 8.3(a). Here is a brief summary of the different types:

- **Slide switch:** A slide switch has two different positions: on and off. The switch is manually moved to one position or the other. For microcontroller applications, slide switches are available

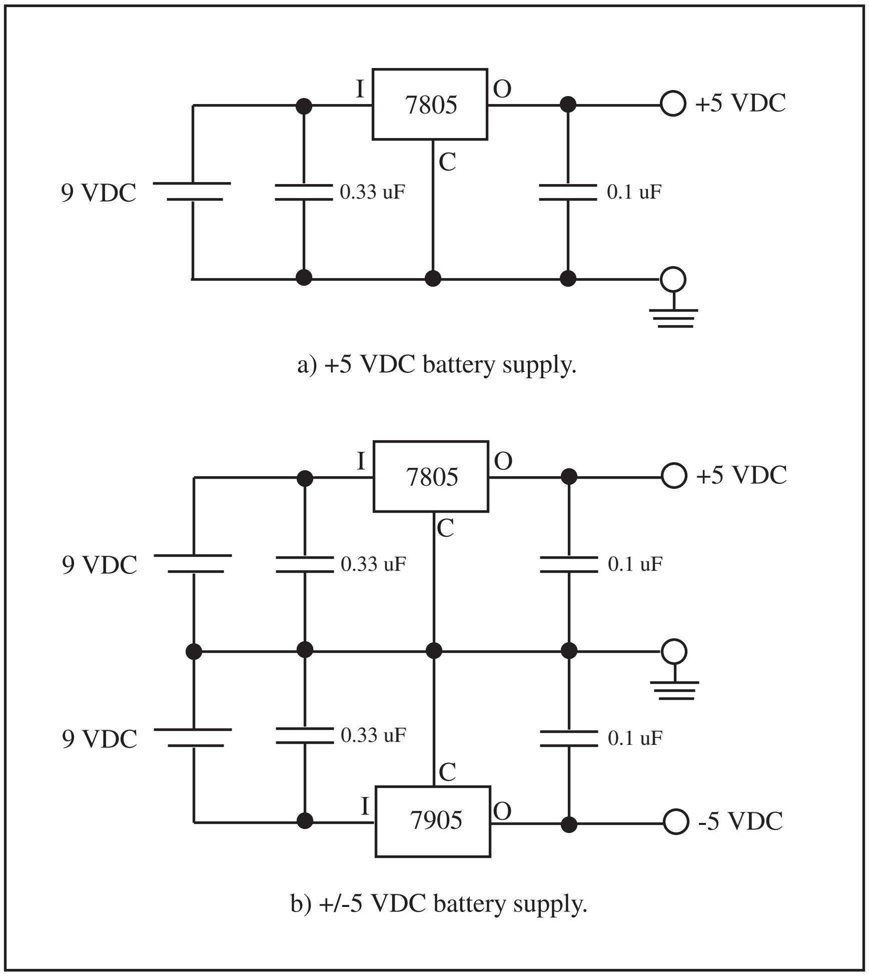


Figure 8.2: Battery supply circuits employing a 9 VDC battery with a 5 VDC regulators.

that fit in the profile of a common integrated circuit size dual inline package (DIP). A bank of four or eight DIP switches in a single package is commonly available.

- **Momentary contact pushbutton switch:** A momentary contact pushbutton switch comes in two varieties: normally closed (NC) and normally open (NO). A normally open switch, as its name implies, does not normally provide an electrical connection between its contacts. When the pushbutton portion of the switch is depressed, the connection between the two switch contacts is made. The connection is held as long as the switch is depressed. When the switch is released the connection is opened. The converse is true for a normally closed switch. For microcontroller applications, pushbutton switches are available in a small tact type switch configuration.

- **Push on/push off switches:** These type of switches are also available in a normally open or normally closed configuration. For the normally open configuration, the switch is depressed to make connection between the two switch contacts. The pushbutton must be depressed again to release the connection.

- **Hexadecimal rotary switches:** Small profile rotary switches are available for microcontroller applications. These switches commonly have sixteen rotary switch positions. As the switch is rotated to each position, a unique four bit binary code is provided at the switch contacts.

A common switch interface is shown in Figure 8.3(b). This interface allows a logic one or zero to be properly introduced to a microcontroller input port pin. The basic interface consists of the switch in series with a current limiting resistor. The node between the switch and the resistor is provided to the microcontroller input pin. In the configuration shown, the resistor pulls the microcontroller input up to the supply voltage V_{DD}. When the switch is closed, the node is grounded and a logic zero is provided to the microcontroller input pin. To reverse the logic of the switch configuration, the position of the resistor and the switch is simply reversed.

8.4.2 PULLUP RESISTORS IN SWITCH INTERFACE CIRCUITRY

Many microcontrollers are equipped with pullup resistors at the input pins. The pullup resistors are asserted with the appropriate register setting. The pullup resistor replaces the external resistor in the switch configuration as shown in Figure 8.3b) right.

8.4.3 SWITCH DEBOUNCING

Mechanical switches do not make a clean transition from one position (on) to another (off). When a switch is moved from one position to another, it makes and breaks contact multiple times. This activity may go on for tens of milliseconds. A microcontroller is relatively fast as compared to the action of the switch. Therefore, the microcontroller is able to recognize each switch bounce as a separate and erroneous transition.

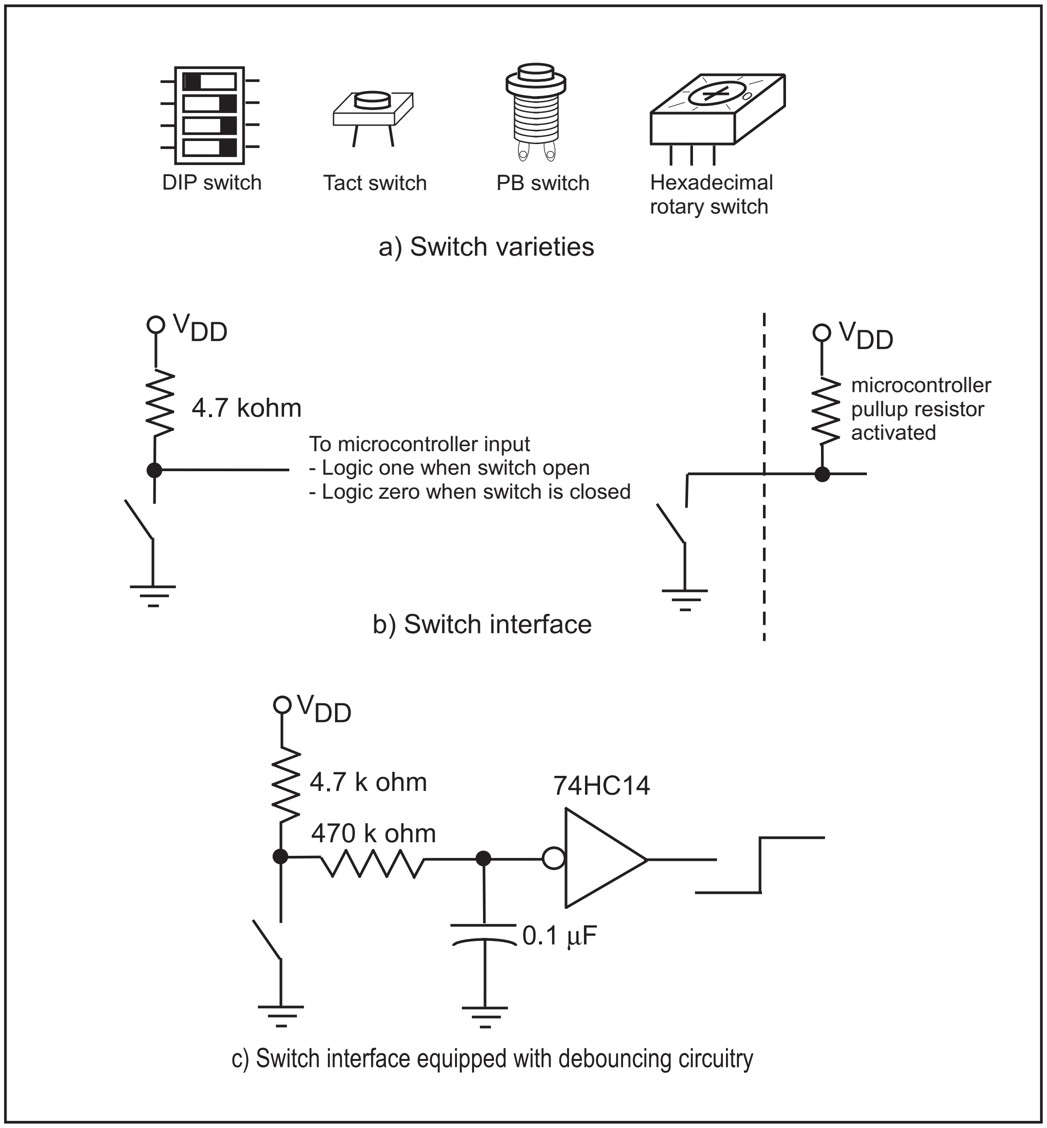

DIP switch
Tact switch
PB switch
Hexadecimal rotary switch
a) Switch varieties
VDD
4.7 kohm
To microcontroller input
- Logic one when switch open
- Logic zero when switch is closed
VDD
microcontroller pullup resistor activated
b) Switch interface
VDD
4.7 k ohm
470 k ohm
74HC14
0.1 µF
c) Switch interface equipped with debouncing circuitry

Figure 8.3: Switch interface.

To correct the switch bounce phenomena additional external hardware components may be used or software techniques may be employed. A hardware debounce circuit is illustrated in Figure 8.3(c). The node between the switch and the limiting resistor of the basic switch circuit is fed to a low pass filter (LPF) formed by the 470 k ohm resistor and the capacitor. The LPF prevents abrupt changes (bounces) in the input signal from the microcontroller. The LPF is followed by a 74HC14 Schmitt Trigger, which is simply an inverter equipped with hysteresis. This further limits the switch bouncing.

Switches may also be debounced using software techniques. This is accomplished by inserting a 30 to 50 ms lockout delay in the function responding to port pin changes. The delay prevents the microcontroller from responding to the multiple switch transitions related to bouncing.

You must carefully analyze a given design to determine if hardware or software switch debouncing techniques will be used. It is important to remember that all switches exhibit bounce phenomena and, therefore, must be debounced.

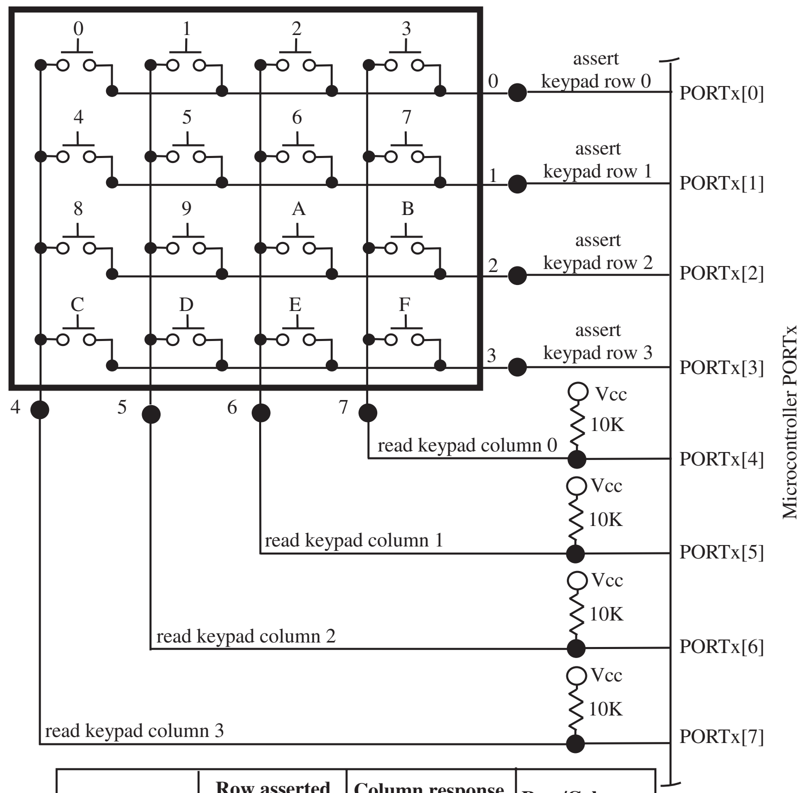
8.4.4 KEYPADS

A keypad is simply an extension of the simple switch configuration. A typical keypad configuration and interface are shown in Figure 8.4. As you can see the keypad is simply multiple switches in the same package. A hexadecimal keypad is provided in the figure. A single row of keypad switches are asserted by the microcontroller and then the host keypad port is immediately read. If a switch has been depressed, the keypad pin corresponding to the column the switch is in will also be asserted. The combination of a row and a column assertion can be decoded to determine which key has been pressed as illustrated in the table. Keypad rows are continually asserted one after the other in sequence. Since the keypad is a collection of switches, debounce techniques must also be employed.

The keypad may be used to introduce user requests to a microcontroller. A standard keypad with alphanumeric characters may be used to provide alphanumeric values to the microcontroller such as providing your personal identification number (PIN) for a financial transaction. However, some keypads are equipped with removable switch covers such that any activity can be associated with a key press.

In Figure 8.5, we have connected the ATmega328 to a hexadecimal keypad via PORTB. PORTB[3:0] is configured as output to selectively assert each row. PORTB[7:4] is configured as input. Each row is sequentially asserted low. Each column is then read via PORTB[7:4] to see if any switch in that row has been depressed. If no switches have been depressed in the row, an "F" will be read from PORTB[7:4]. If a switch has been depressed, some other value than "F" will be read. The read value is then passed into a switch statement to determine the ASCII equivalent of the depressed switch. The function is not exited until the switch is released. This prevents a switch "double hit."

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



Key pressed by user	Row asserted by microcontroller				Column response from keypad switch				Row/Column combination read at micro port
	3	2	1	0	7	6	5	4	
0	1	1	1	0	1	1	1	0	0xEE
1	1	1	1	0	1	1	0	1	0xDE
2	1	1	1	0	1	0	1	1	0xBE
3	1	1	1	0	0	1	1	1	0x7E
4	1	1	0	1	1	1	1	0	0xED
5	1	1	0	1	1	1	0	1	0xDD
6	1	1	0	1	1	0	1	1	0xBD
7	1	1	0	1	0	1	1	1	0x7D
8	1	0	1	1	1	1	1	0	0xEB
9	1	0	1	1	1	1	0	1	0xDB
A	1	0	1	1	1	0	1	1	0xBB
B	1	0	1	1	0	1	1	1	0x7B
C	0	1	1	1	1	1	1	0	0xE7
D	0	1	1	1	1	1	0	1	0xD7
E	0	1	1	1	1	0	1	1	0xB7
F	0	1	1	1	0	1	1	1	0x77
none	X	X	X	X	1	1	1	1	0xXF

Figure 8.4: Keypad interface.

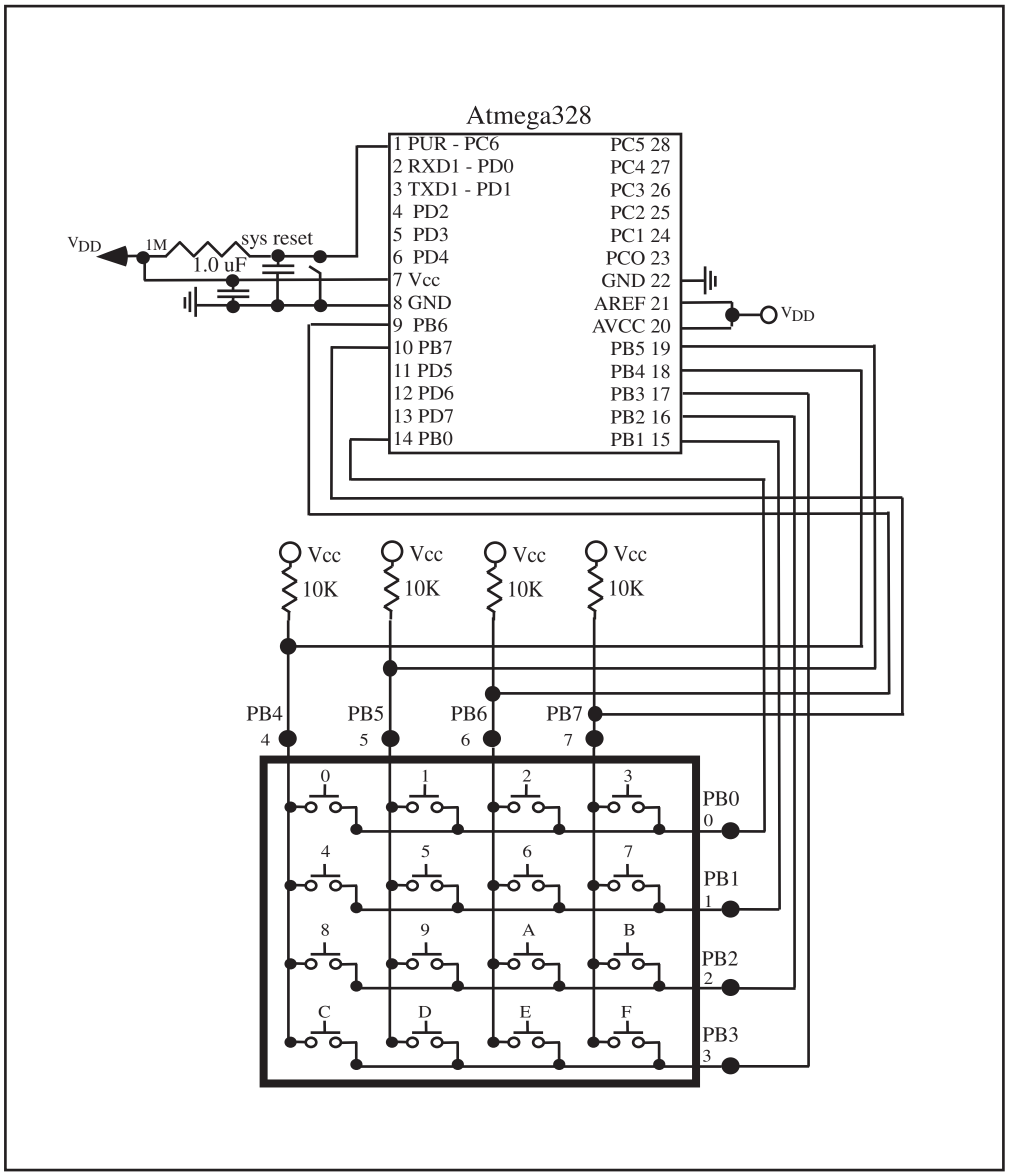


Figure 8.5: Hexadecimal keypad interface to microcontroller.

```
unsigned char get_keypad_value(void)
{
unsigned char  PORTB_value, PORTB_value_masked;
unsigned char  ascii_value;

DDRC = 0x0F;
       //set PORTB[7:4] to input,
                                             //PORTB[3:0] to output

                                             //switch depressed in row 0?
PORTB = 0xFE;                                //assert row 0 via PORTB[0]
PORTB_value = PINB;                          //read PORTB
PORTB_value_masked = (PORTB_value & 0xf0);   //mask PORTB[3:0]

                                             //switch depressed in row 1?
if(PORTB_value_masked == 0xf0)
       //no switches depressed in row 0
   {
   PORTB = 0xFD;                             //assert Row 1 via PORTB[1]
   PORTB_value = PINB;                       //read PORTB
   PORTB_value_masked = (PORTB_value & 0xf0);//mask PORTB[3:0]
   }

                                             //switch depressed in row 2?
if(PORTB_value_masked == 0xf0)
       //no switches depressed in row 0
   {
   PORTB = 0xFB;                             //assert Row 2 via PORTC[2]
   PORTB_value = PINB;                       //read PORTB
   PORTB_value_masked = (PORTB_value & 0xf0);//mask PORTB[3:0]
   }

                                             //switch depressed in row 3?
if(PORTB_value_masked == 0xf0)
       //no switches depressed in row 0
   {
   PORTB = 0xF7;                             //assert Row 3 via PORTB[3]
   PORTB_value = PINB;                       //read PORTB
   PORTB_value_masked = (PORTB_value & 0xf0);//mask PORTB[3:0]
   }

if(PORTB_value_masked != 0xf0)
  {
  switch(PORTB_value_masked)
    {
    case 0xEE: ascii_value = '0';
                break;

    case 0xDE: ascii_value = '1';
               break;

    case 0xBE: ascii_value = '2';
               break;

    case 0x7E: ascii_value = '3';
               break;

    case 0xED: ascii_value = '4';
               break;

    case 0xDD: ascii_value = '5';
               break;

    case 0xBD: ascii_value = '6';
               break;

    case 0x7D: ascii_value = '7';
               break;

    case 0xEB: ascii_value = '8';
               break;

    case 0xDB: ascii_value = '9';
               break;

    case 0xBB: ascii_value = 'a';
               break;
```

```
    case 0x&B: ascii_value = 'b';
               break;

    case 0xE7: ascii_value = 'c';
               break;

    case 0xD7: ascii_value = 'd';
               break;

    case 0xB7: ascii_value = 'e';
               break;

    case 0x77: ascii_value = 'f';
               break;

    default:;
    }

while(PORTB_value_masked != 0xf0);
       //wait for key to be released

return ascii_value;
}

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

8.4.5 SENSORS

A microcontroller is typically used in control applications where data is collected, the data is assimilated and processed by the host algorithm, and a control decision and accompanying signals are provided by the microcontroller. Input data for the microcontroller is collected by a complement of input sensors. These sensors may be digital or analog in nature.

8.4.5.1 Digital Sensors

Digital sensors provide a series of digital logic pulses with sensor data encoded. The sensor data may be encoded in any of the parameters associated with the digital pulse train such as duty cycle, frequency, period, or pulse rate. The input portion of the timing system may be configured to measure these parameters.

An example of a digital sensor is the optical encoder. An optical encoder consists of a small plastic transparent disk with opaque lines etched into the disk surface. A stationary optical emitter and detector pair is placed on either side of the disk. As the disk rotates, the opaque lines break the continuity between the optical source and detector. The signal from the optical detector is monitored to determine disk rotation as shown in Figure 8.6.

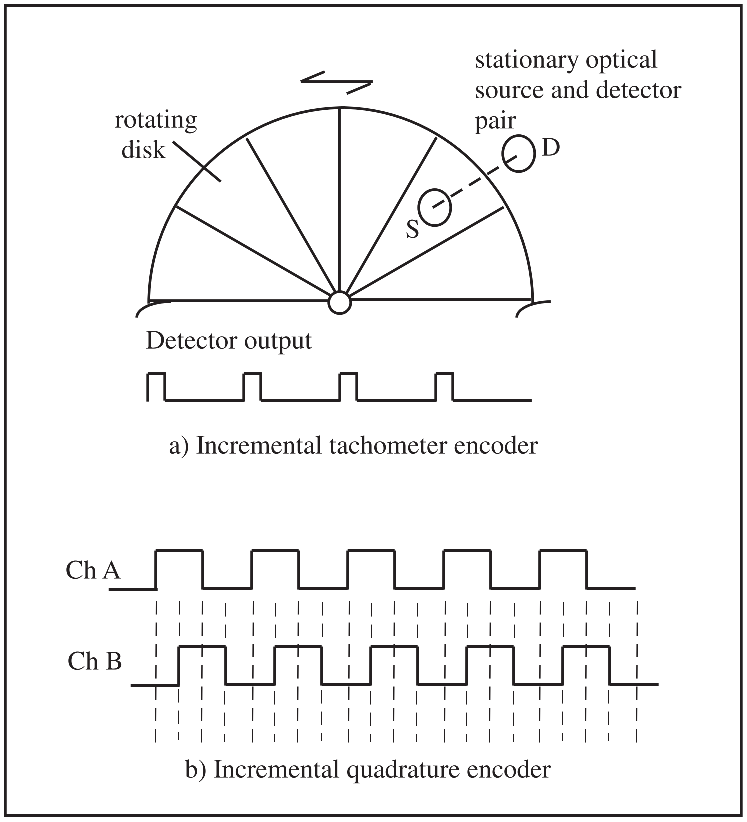


Figure 8.6: Optical encoder.

Optical encoders are available in a variety of types depending on the information desired. There are two major types of optical encoders: incremental encoders and absolute encoders. An absolute encoder is used when it is required to retain position information when power is lost. For example, if you were using an optical encoder in a security gate control system, an absolute encoder would be used to monitor the gate position. An incremental encoder is used in applications where a velocity or a velocity and direction information is required.

The incremental encoder types may be further subdivided into tachometers and quadrature encoders. An incremental tachometer encoder consists of a single track of etched opaque lines as shown in Figure 8.6(a). It is used when the velocity of a rotating device is required. To calculate velocity, the number of detector pulses are counted in a fixed amount of time. Since the number of pulses per encoder revolution is known, velocity may be calculated.

The quadrature encoder contains two tracks shifted in relationship to one another by 90 degrees. This allows the calculation of both velocity and direction. To determine direction, one would monitor the phase relationship between Channel A and Channel B as shown in Figure 8.6(b). The absolute encoder is equipped with multiple data tracks to determine the precise location of the encoder disk [Sick Stegmann].

8.4.5.2 Analog Sensors

Analog sensors provide a DC voltage that is proportional to the physical parameter being measured. As discussed in the analog to digital conversion chapter, the analog signal may be first preprocessed by external analog hardware such that it falls within the voltage references of the conversion subsystem. The analog voltage is then converted to a corresponding binary representation.

An example of an analog sensor is the flex sensor shown in Figure 8.7(a). The flex sensor provides a change in resistance for a change in sensor flexure. At 0 degrees flex, the sensor provides 10k ohms of resistance. For 90 degrees flex, the sensor provides 30-40k ohms of resistance. Since the microcontroller can not measure resistance directly, the change in flex sensor resistance must be converted to a change in a DC voltage. This is accomplished using the voltage divider network shown in Figure 8.7(c). For increased flex, the DC voltage will increase. The voltage can be measured using the ATmega328's analog to digital converter subsystem. The flex sensor may be used in applications such as virtual reality data gloves, robotic sensors, biometric sensors, and in science and engineering experiments [Images Company].

8.4.6 LM34 TEMPERATURE SENSOR EXAMPLE

Temperature may be sensed using an LM34 (Fahrenheit) or LM35 (Centigrade) temperature transducer. The LM34 provides an output voltage that is linearly related to temperature. For example, the LM34D operates from 32 degrees F to 212 degrees F providing +10mV/degree Fahrenheit resolution with a typical accuracy of ± 0.5 degrees Fahrenheit [National]. This sensor is used in the automated cooling fan example at the end of the chapter. The output from the sensor is typically connected to the ADC input of the microcontroller.

8.5 OUTPUT DEVICES

As previously mentioned, an external device should not be connected to a microcontroller without first performing careful interface analysis to ensure the voltage, current, and timing requirements of the microcontroller and the external device. In this section, we describe interface considerations for a wide variety of external devices. We begin with the interface for a single light emitting diode.

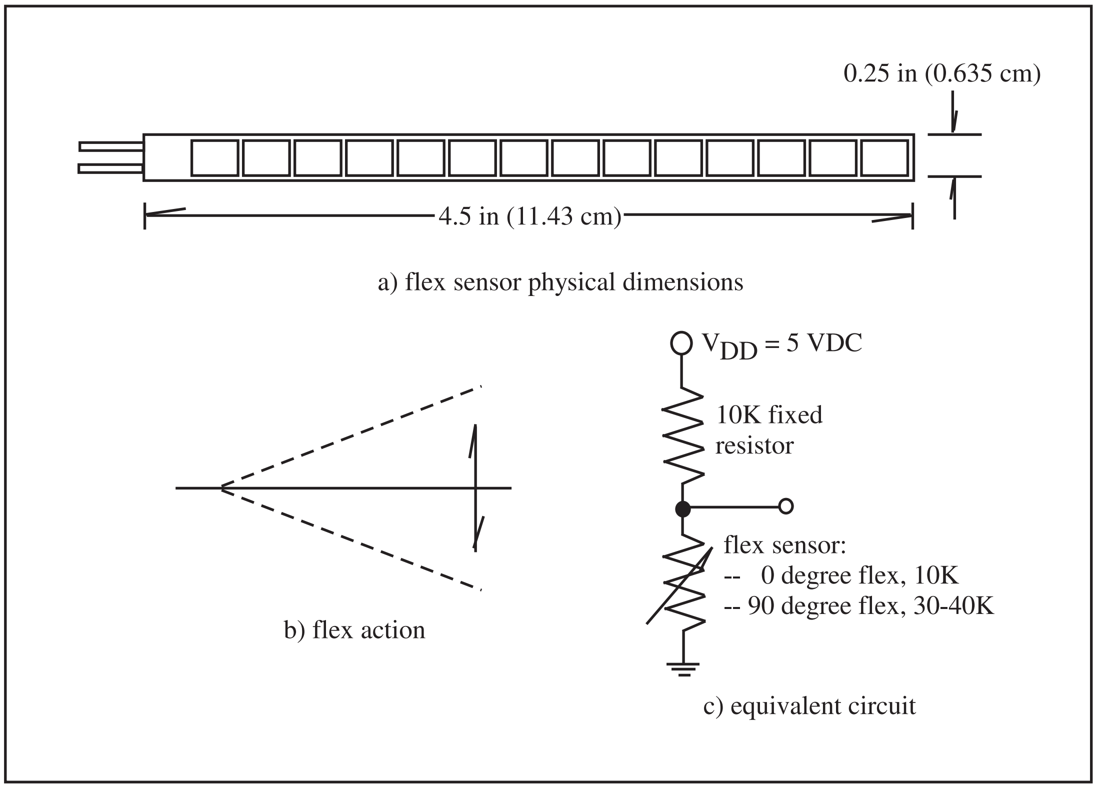


Figure 8.7: Flex sensor.

8.5.1 LIGHT EMITTING DIODES (LEDS)

An LED is typically used as a logic indicator to inform the presence of a logic one or a logic zero at a specific pin of a microcontroller. An LED has two leads: the anode or positive lead and the cathode or negative lead. To properly bias an LED, the anode lead must be biased at a level approximately 1.7 to 2.2 volts higher than the cathode lead. This specification is known as the forward voltage (V_f) of the LED. The LED current must also be limited to a safe level known as the forward current (I_f). The diode voltage and current specifications are usually provided by the manufacturer.

An example of an LED biasing circuit is provided in Figure 8.8. A logic one is provided by the microcontroller to the input of the inverter. The inverter provides a logic zero at its output which provides a virtual ground at the cathode of the LED. Therefore, the proper voltage biasing for the LED is provided. The resistor (R) limits the current through the LED. A proper resistor value can be calculated using $R = (V_{DD} - V_{DIODE})/I_{DIODE}$. It is important to note that a 7404 inverter must be used due to its capability to safely sink 16 mA of current. Alternately, an NPN transistor such as a 2N2222 (PN2222 or MPQ2222) may be used in place of the inverter as shown in the figure. In Chapter 1, we used large (10 mm) red LEDs in the KNH instrumentation project. These LEDs have V_f of 6 to 12 VDC and I_f of 20 mA at 1.85 VDC. This requires the interface circuit shown in Figure 8.8c) right.

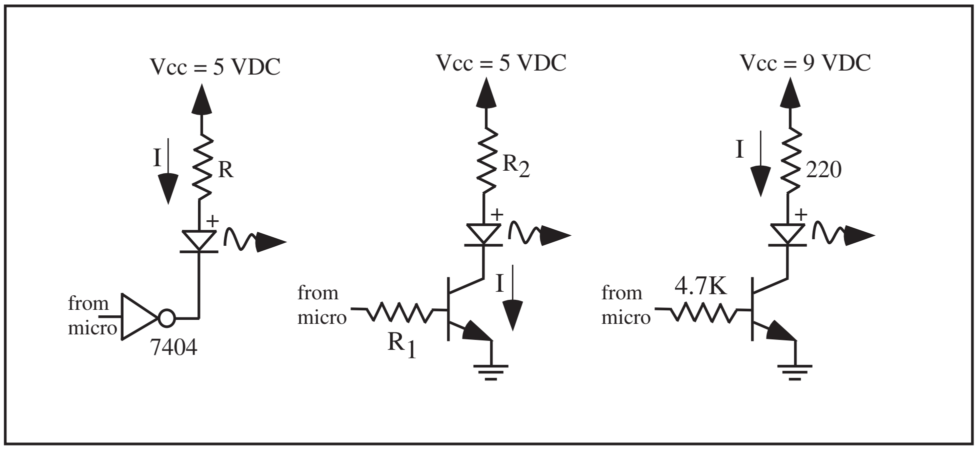


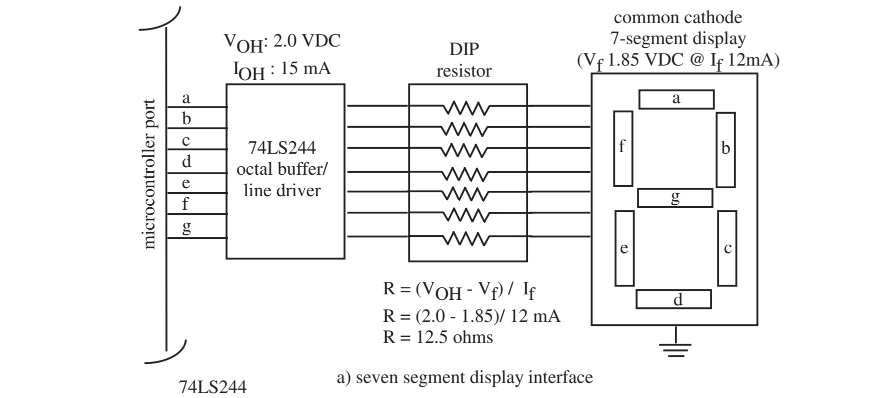
Figure 8.8: LED display devices.

8.5.2 SEVEN SEGMENT LED DISPLAYS

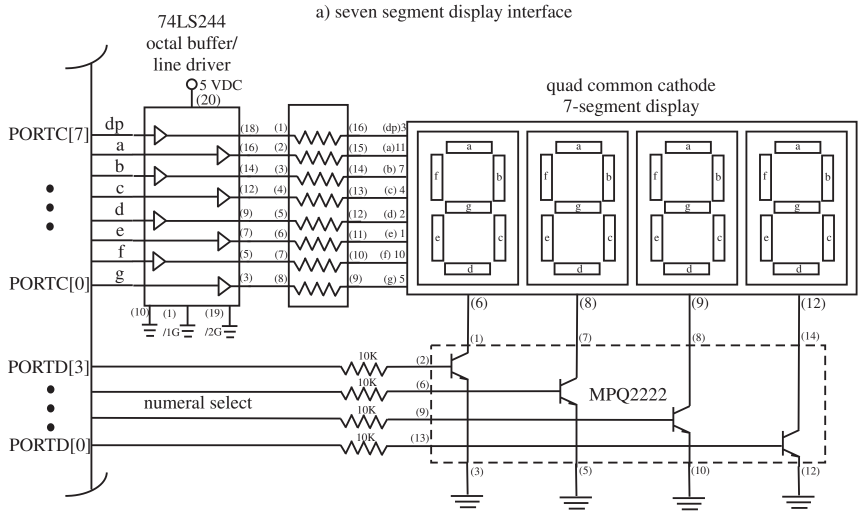
To display numeric data, seven segment LED displays are available as shown in Figure 8.9(a). Different numerals can be displayed by asserting the proper LED segments. For example, to display the number five, segments a, c, d, f, and g would be illuminated. Seven segment displays are available in common cathode (CC) and common anode (CA) configurations. As the CC designation implies, all seven individual LED cathodes on the display are tied together.

The microcontroller is not capable of driving the LED segments directly. As shown in Figure 8.9(a), an interface circuit is required. We use a 74LS244 octal buffer/driver circuit to boost the current available for the LED. The LS244 is capable of providing 15 mA per segment (I_{OH}) at 2.0 VDC (V_{OH}). A limiting resistor is required for each segment to limit the current to a safe value for the LED. Conveniently, resistors are available in DIP packages of eight for this type of application.

Seven segment displays are available in multi-character panels. In this case, separate microcontroller ports are not used to provide data to each seven segment character. Instead, a single port is used to provide character data. A portion of another port is used to sequence through each of the characters as shown in Figure 8.9(b). An NPN (for a CC display) transistor is connected to the common cathode connection of each individual character. As the base contact of each transistor is sequentially asserted, the specific character is illuminated. If the microcontroller sequences through the display characters at a rate greater than 30 Hz, the display will have steady illumination.



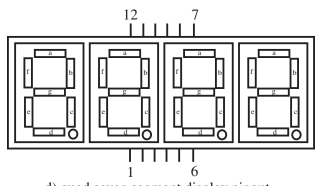
a) seven segment display interface



b) quad seven segment display interface

numeral	dp PORTC[7]	a PORTC[6]	b PORTC[5]	c PORTC[4]	d PORTC[3]	e PORTC[2]	f PORTC[1]	g PORTC[0]	hex rep
0		1	1	1	1	1	1	0	0x7E
1		0	1	1	0	0	0	0	0x30
2		1	1	0	1	1	0	1	0x6D
3		1	1	1	1	0	0	1	0x79
4		0	1	1	0	0	1	1	0x33
5		1	0	1	1	0	1	1	0x5D
6		0	0	1	1	1	1	1	0x1F
7		1	1	1	0	0	0	0	0x70
8		1	1	1	1	1	1	1	0x7F
9		1	1	1	0	0	1	1	0x73

c) numeral to segment converion



d) quad seven segment display pinout
UN(M)5624-11 EWRS

Figure 8.9: Seven segment LED display devices.

8.5.3 CODE EXAMPLE

Provided below is a function used to illuminate the correct segments on a multi-numeral seven display. The numeral is passed in as an argument to the function along with the numerals position on the display and also an argument specifying whether or not the decimal point (dp) should be displayed at that position. The information to illuminate specific segments are provided in Figure 8.9c).

```
//***************************************************************************
void LED_character_display(unsigned int numeral, unsigned int position,
                           unsigned int decimal_point)
{
unsigned char output_value;

                                       //illuminate numerical segments
switch(numeral)
  {
  case 0: output_value = 0x7E;
    break;

  case 1: output_value = 0x30;
    break;

  case 2: output_value = 0x6D;
    break;

  case 3: output_value = 0x79;
    break;

  case 4: output_value = 0x33;
    break;

  case 5: output_value = 0x5D;
    break;

  case 6: output_value = 0x1F;
    break;

  case 7: output_value = 0x70;
    break;
  case 8: output_value = 0x7F;
    break;

  case 9: output_value = 0x73;
    break;

  default:;
  }

if(decimal_point != 0)
  PORTB = output_value | 0x80;         //illuminate decimal point

switch(position)                       //assert position
  {
  case 0: PORTD = 0x01;                //least significant bit
          break;

  case 1: PORTD = 0x02;                //least significant bit + 1
          break;

  case 2: PORTD = 0x04;                //least significant bit + 2
          break;

  case 3: PORTD = 0x08;                //most significant bit
          break;

  default:;
  }
}
//*************************************************************************
```

8.5.4 TRI-STATE LED INDICATOR

The tri-state LED indicator is shown in Figure 8.10. It is used to provide the status of an entire microcontroller port. The indicator bank consists of eight green and eight red LEDs. When an individual port pin is logic high, the green LED is illuminated. When logic low, the red LED is illuminated. If the port pin is at a tri-state high impedance state, no LED is illuminated.

The NPN/PNP transistor pair at the bottom of the figure provides a 2.5 VDC voltage reference for the LEDs. When a specific port pin is logic high (5.0 VDC), the green LED will be forward biased since its anode will be at a higher potential than its cathode. The 47 ohm resistor limits

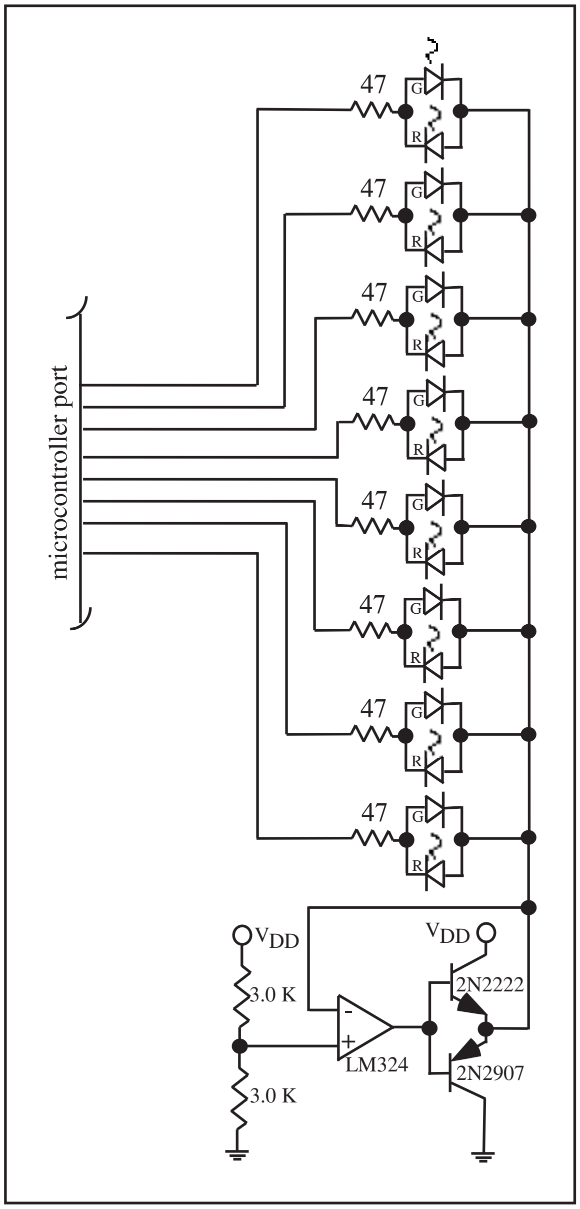


Figure 8.10: Tri-state LED display.

current to a safe value for the LED. Conversely, when a specific port pin is at a logic low (0 VDC) the red LED will be forward biased and illuminate. For clarity, the red and green LEDs are shown as being separate devices. LEDs are available that have both LEDs in the same device.

8.5.5 DOT MATRIX DISPLAY

The dot matrix display consists of a large number of LEDs configured in a single package. A typical 5 x 7 LED arrangement is a matrix of five columns of LEDs with seven LEDs per row as shown in Figure 8.11. Display data for a single matrix column [R6-R0] is provided by the microcontroller. That specific row is then asserted by the microcontroller using the column select lines [C2-C0]. The entire display is sequentially built up a column at a time. If the microcontroller sequences through each column fast enough (greater than 30 Hz), the matrix display appears to be stationary to a human viewer.

In Figure 8.11(a), we have provided the basic configuration for the dot matrix display for a single display device. However, this basic idea can be expanded in both dimensions to provide a multi-character, multi-line display. A larger display does not require a significant number of microcontroller pins for the interface. The dot matrix display may be used to display alphanumeric data as well as graphics data. In Figure 8.11(b), we have provided additional detail of the interface circuit.

8.5.6 LIQUID CRYSTAL CHARACTER DISPLAY (LCD) IN C

An LCD is an output device to display text information as shown in Figure 8.12. LCDs come in a wide variety of configurations, including multi-character, multi-line format. A 16 x 2 LCD format is common. That is, it has the capability of displaying two lines of 16 characters each. The characters are sent to the LCD via American Standard Code for Information Interchange (ASCII) format a single character at a time. For a parallel configured LCD, an eight bit data path and two lines are required between the microcontroller and the LCD. A small microcontroller mounted to the back panel of the LCD translates the ASCII data characters and control signals to properly display the characters. LCDs are configured for either parallel or serial data transmission format. In the example provided, we use a parallel configured display.

8.5.6.1 Programming an LCD in C

Some sample C code is provided below to send data and control signals to an LCD. In this specific example, an AND671GST 1 x 16 character LCD was connected to the Atmel ATmega328 microcontroller. One 8-bit port and two extra control lines are required to connect the microcontroller to the LCD. Note: The initialization sequence for the LCD is specified within the manufacturer's technical data.

```
//*********************************************************************
//Internal Oscillator: 1 MHz
//ATMEL AVR ATmega328
//Chip Port Function I/O Source/Dest Asserted Notes
```

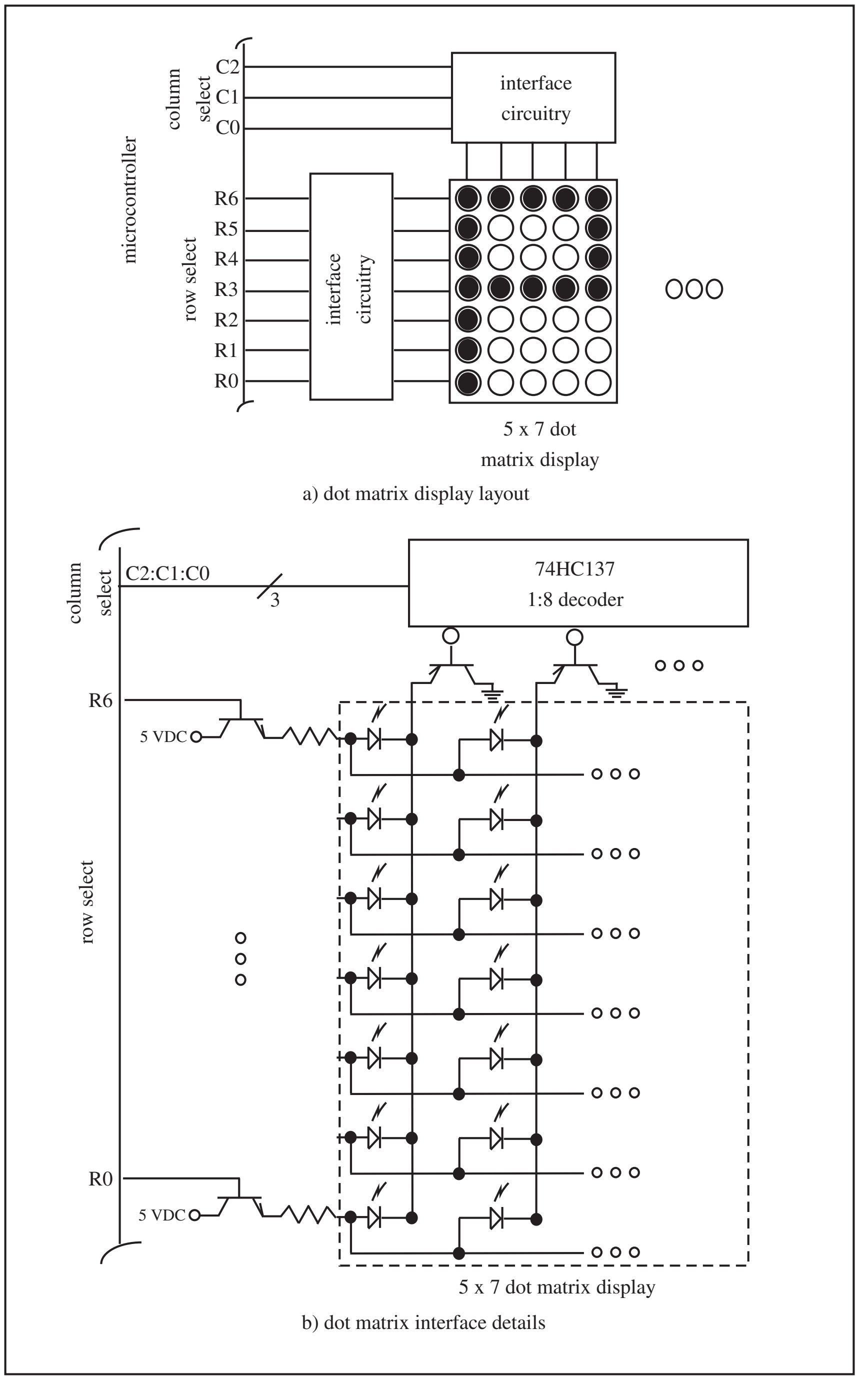


Figure 8.11: Dot matrix display.

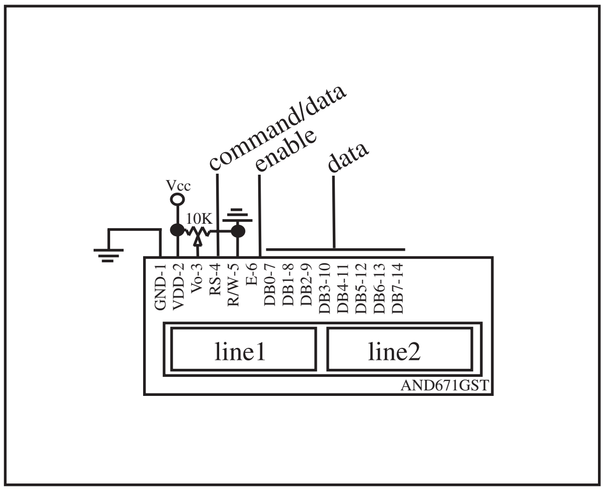


Figure 8.12: LCD display.

```
//************************************************************************
//Pin 1: /Reset
//Pin 2: PD0 to DB0 LCD
//Pin 3: PD1 to DB1 LCD
//Pin 4: PD2 to DB2 LCD
//Pin 5: PD3 to DB3 LCD
//Pin 6: PD4 to DB4 LCD
//Pin 7: Vcc
//Pin 8: Gnd
//Pin 11: PD5 to DB6 LCD
//Pin 12: PD6 to DB6 LCD
//Pin 13: PD7 to DB7 LCD
//Pin 20: AVCC to Vcc
//Pin 21: AREF to Vcc
//Pin 22 Gnd
//Pin 27 PC4 to LCD Enable (E)
//Pin 28 PC5 to LCD RS

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

//ATMEL register definitions for ATmega328
#include<iom328v.h>
```

```
//function prototypes*****************************************************
void delay(unsigned int number_of_65_5ms_interrupts);
void init_timer0_ovf_interrupt(void);
void initialize_ports(void);              //initializes ports
void power_on_reset(void);                //returns system to startup state
void clear_LCD(void);                     //clears LCD display
void LCD_Init(void);                      //initialize AND671GST LCD
void putchar(unsigned char c);            //send character to LCD
void putcommand(unsigned char c);         //send command to LCD
void timer0_interrupt_isr(void);
void perform_countdown(void);
void clear_LCD(void);
void systems_A_OK(void);
void print_mission_complete(void);
void convert_int_to_string_display_LCD(unsigned int total_integer_value);

//program constants
#define TRUE     1
#define FALSE     0
#define OPEN     1
#define CLOSE     0
#define YES     1
#define NO     0
#define SAFE     1
#define UNSAFE     0
#define ON           1
#define OFF          0

//interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:17

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

//global variables

void main(void)
{
```

```
init_timer0_ovf_interrupt();
//initialize Timer0 to serve as elapsed
initialize_ports();                   //initialize ports

perform_countdown();
delay(46);
:
:
:
systems_A_OK();

}//end main

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

//*************************************************************************
//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 = 0xff;                //PORTC[7:0] as output
PORTC= 0x00;                //initialize low

DDRD = 0xff;                //PORTB[7:0] as output
PORTD= 0x00;                //initialize low
}

//*************************************************************************
//delay(unsigned int num_of_65_5ms_interrupts): this generic delay function
//provides the specified delay as the number of 65.5 ms "clock ticks" from
//the Timer0 interrupt.
//Note: this function is only valid when using a 1 MHz crystal or ceramic
//      resonator
//*************************************************************************

void delay(unsigned int number_of_65_5ms_interrupts)
```

```
{
TCNT0 = 0x00;                         //reset timer0
delay_timer = 0;
while(delay_timer <= number_of_65_5ms_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 1 MHz and then
//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)
{
TCCR0 = 0x04; //divide timer0 timebase by 256, overflow occurs every 65.5ms
TIMSK = 0x01; //enable timer0 overflow interrupt
asm("SEI");   //enable global interrupt
}

//**************************************************************************
//LCD_Init: initialization for an LCD connected in the following manner:
//LCD: AND671GST 1x16 character display
//LCD configured as two 8 character lines in a 1x16 array
//LCD data bus (pin 14-pin7) ATMEL 8: PORTD
//LCD RS (pin 28)  ATMEL 8: PORTC[5]
//LCD E  (pin 27)  ATMEL 8: PORTC[4]
//**************************************************************************

void LCD_Init(void)
{
delay(1);
delay(1);
delay(1);
                   // output command string to initialize LCD
```

```
putcommand(0x38);  //function set 8-bit
delay(1);
putcommand(0x38);  //function set 8-bit
putcommand(0x38);  //function set 8-bit
putcommand(0x38);  //one line, 5x7 char
putcommand(0x0C);  //display on
putcommand(0x01);  //display clear-1.64 ms
putcommand(0x06);  //entry mode set
putcommand(0x00);  //clear display, cursor at home
putcommand(0x00);  //clear display, cursor at home
}

//*********************************************************************
//putchar:prints specified ASCII character to LCD
//*********************************************************************

void putchar(unsigned char c)
{
DDRD  = 0xff;                            //set PORTD as output
DDRC  = DDRC|0x30;                      //make PORTC[5:4] output
PORTD = c;
PORTC = (PORTC|0x20)|PORTC_pullup_mask;   //RS=1
PORTC = (PORTC|0x10)|PORTC_pullup_mask;;  //E=1
PORTC = (PORTC&0xef)|PORTC_pullup_mask;;  //E=0
delay(1);
}

//*********************************************************************
//putcommand: performs specified LCD related command
//*********************************************************************

void putcommand(unsigned char d)
{
DDRD  = 0xff;                            //set PORTD as output
DDRC  = DDRC|0xC0;                      //make PORTA[5:4] output
PORTC = (PORTC&0xdf)|PORTC_pullup_mask;   //RS=0
PORTD = d;
PORTC = (PORTC|0x10)|PORTC_pullup_mask;   //E=1
PORTC = (PORTC&0xef)|PORTC_pullup_mask;   //E=0
```

```
delay(1);
}

//*************************************************************************
//clear_LCD: clears LCD
//*************************************************************************

void clear_LCD(void)
{
putcommand(0x01);
}

//*************************************************************************
//void timer0_interrupt_isr(void)
//*************************************************************************

void timer0_interrupt_isr(void)
{
delay_timer++;
}
//*************************************************************************
//void perform_countdown(void)
//*************************************************************************

void perform_countdown(void)
{
clear_LCD();
putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('1'); putchar ('0'); //print 10
delay(15);                   //delay 1s

putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('9');                //print 9
delay(15);                   //delay 1s

putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
```

```
putchar('8');                //print 8
delay(15);                   //delay 1s

putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('7');                //print 7
delay(15);                   //delay 1s

putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('6');                //print 6
delay(15);                   //delay 1s

putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('5');                //print 5
delay(15);                   //delay 1s

putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('4');                //print 4
delay(15);                   //delay 1s

putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('3');                //print 3
delay(15);                   //delay 1s

putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('2');                //print 2
delay(15);                   //delay 1s

putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('1');                //print 1
delay(15);                   //delay 1s
```

```
putcommand(0x01);              //cursor home
putcommand(0x80);              //DD RAM location 1 - line 1
putchar('0');                  //print 0
delay(15);                     //delay 1s

//BLASTOFF!
putcommand(0x01);              //cursor home

putcommand(0x80);              //DD RAM location 1 - line 1
putchar('B'); putchar('L'); putchar('A'); putchar('S'); putchar('T');
putchar('O'); putchar('F'); putchar('F'); putchar('!');

}

//***************************************************************************
//void systems_A_OK(void)
//***************************************************************************

void systems_A_OK(void)
{
clear_LCD();
putcommand(0x01);              //cursor home
putcommand(0x80);              //DD RAM location 1 - line 1
putchar('S'); putchar('Y'); putchar('S'); putchar('T'); putchar('E');
putchar('M'); putchar('S'); putchar(' '); putchar('A'); putchar('-');
putchar('O'); putchar('K'); putchar('!'); putchar('!'); putchar('!');
}

//***************************************************************************
//void print_mission_complete(void)
//***************************************************************************

void print_mission_complete(void)
{
clear_LCD();
putcommand(0x01);              //cursor home
putcommand(0x80);              //DD RAM location 1 - line 1
putchar('M'); putchar('I'); putchar('S'); putchar('S'); putchar('I');
putchar('O'); putchar('N');
```

```
putcommand(0xC0);//DD RAM location 1 - line 2
putchar('C'); putchar('O'); putchar('M'); putchar('P'); putchar('L');
putchar('E'); putchar('T'); putchar('E'); putchar('!');
}

//*************************************************************************
//end of file

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

8.5.7 LIQUID CRYSTAL CHARACTER DISPLAY (LCD) USING THE ARDUINO DEVELOPMENT ENVIRONMENT

The Arduino Development Environment provides full support to interface an Arduino Duemilanove to either a 4-bit or 8-bit configured LCD. In the 4-bit mode precious output pins are preserved for other uses over the 8-bit configuration [www.arduino.cc].

The Arduino Development Environment hosts the following LCD related functions:

- **Hello World:** This function displays the greeting "hello world" and also displays elapsed time since the last reset.
- **Blink:** The Blink function provides cursor control.
- **Cursor:** This function provides control over the underscore style cursor.
- **Display:** This function blanks the display without the loss of displayed information.
- **Text Direction:** This function controls which direction text flows from the cursor.
- **Autoscroll:** This function provides automatic scrolling to the new text.
- **Serial Input:** This function accepts serial input and displays it on the LCD.
- **SetCursor:** This function sets the cursor position on the LCD.
- **Scroll:** This function scrolls text left and right.

Rather than include the excellent Arduino resource here, the interested reader is referred to the Arduino website for sample code and full documentation [www.arduino.cc].

8.5.8 HIGH POWER DC DEVICES

A number of direct current devices may be controlled with an electronic switching device such as a MOSFET. Specifically, an N-channel enhancement MOSFET (metal oxide semiconductor field effect transistor) may be used to switch a high current load on and off (such as a motor) using a low current control signal from a microcontroller as shown in Figure 8.13(a). The low current control signal from the microcontroller is connected to the gate of the MOSFET. The MOSFET switches the high current load on and off consistent with the control signal. The high current load is connected between the load supply and the MOSFET drain. It is important to note that the load supply voltage and the microcontroller supply voltage do not have to be at the same value. When the control signal on the MOSFET gate is logic high, the load current flows from drain to source. When the control signal applied to the gate is logic low, no load current flows. Thus, the high power load is turned on and off by the low power control signal from the microcontroller.

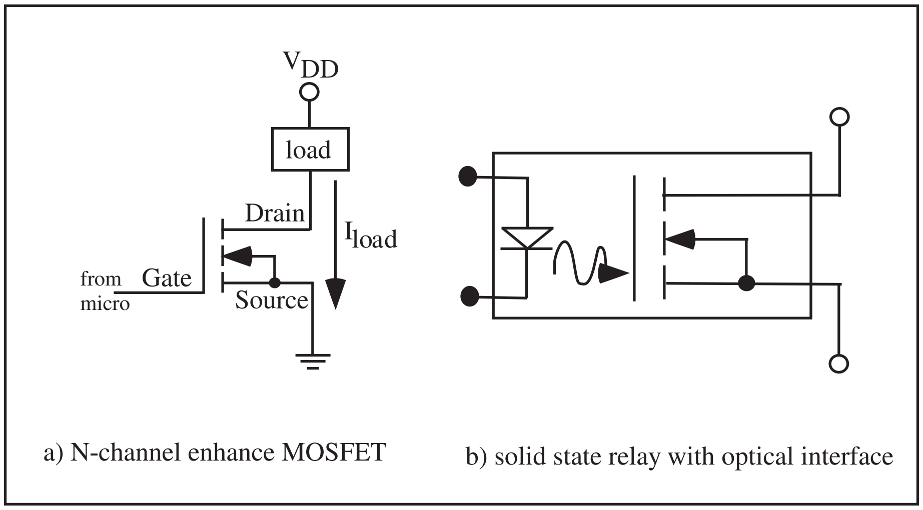

V_{DD}
load
Drain
I_{load}
from micro
Gate
Source
a) N-channel enhance MOSFET
b) solid state relay with optical interface

Figure 8.13: MOSFET circuits.

Often the MOSFET is used to control a high power motor load. A motor is a notorious source of noise. To isolate the microcontroller from the motor noise an optical isolator may be used as an interface as shown in Figure 8.13(b). The link between the control signal from the microcontroller to the high power load is via an optical link contained within a Solid State Relay (SSR). The SSR is properly biased using techniques previously discussed.

8.6 DC SOLENOID CONTROL

The interface circuit for a DC solenoid is provided in Figure 8.14. A solenoid provides a mechanical insertion (or extraction) when asserted. In the interface, an optical isolator is used between the microcontroller and the MOSFET used to activate the solenoid. A reverse biased diode is placed across the solenoid. Both the solenoid power supply and the MOSFET must have the appropriate voltage and current rating to support the solenoid requirements.

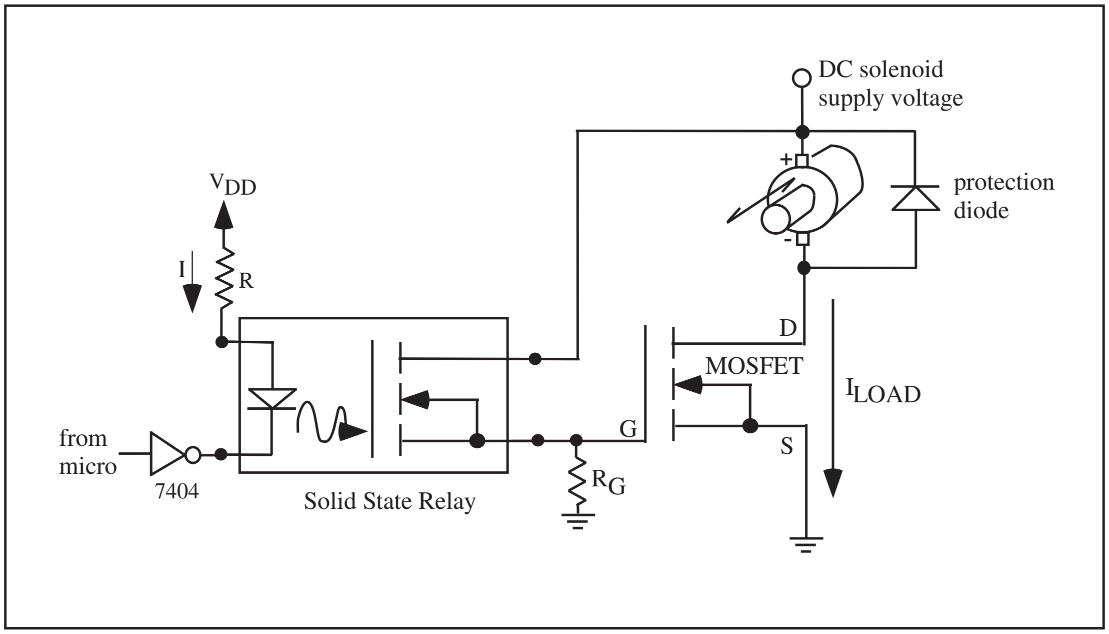


Figure 8.14: Solenoid interface circuit.

8.7 DC MOTOR SPEED AND DIRECTION CONTROL

Often, a microcontroller is used to control a high power motor load. To properly interface the motor to the microcontroller, we must be familiar with the different types of motor technologies. Motor types are illustrated in Figure 8.15.

- **DC motor:** A DC motor has a positive and negative terminal. When a DC power supply of suitable current rating is applied to the motor it will rotate. If the polarity of the supply is switched with reference to the motor terminals, the motor will rotate in the opposite direction. The speed of the motor is roughly proportional to the applied voltage up to the rated voltage of the motor.

- **Servo motor:** A servo motor provides a precision angular rotation for an applied pulse width modulation duty cycle. As the duty cycle of the applied signal is varied, the angular displacement of the motor also varies. This type of motor is used to change mechanical positions such as the steering angle of a wheel.

- **Stepper motor:** A stepper motor as its name implies provides an incremental step change in rotation (typically 2.5 degree per step) for a step change in control signal sequence. The motor is typically controlled by a two or four wire interface. For the four wire stepper motor, the

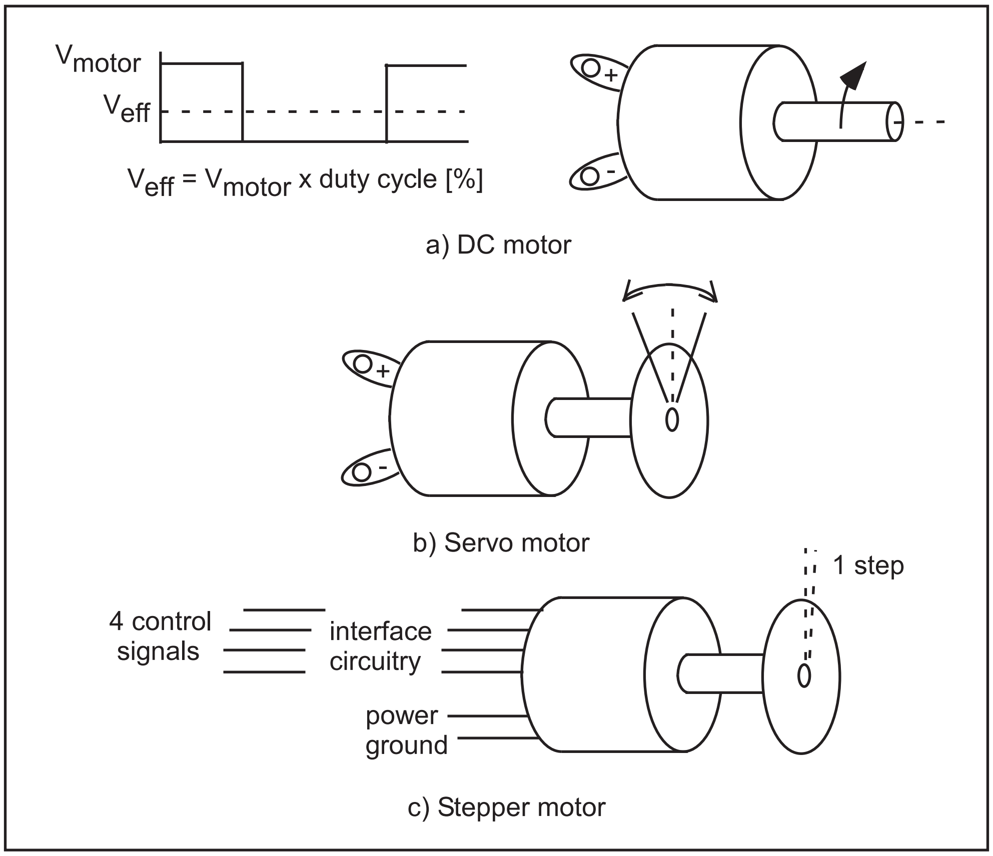


Figure 8.15: Motor types.

microcontroller provides a four bit control sequence to rotate the motor clockwise. To turn the motor counterclockwise, the control sequence is reversed. The low power control signals are interfaced to the motor via MOSFETs or power transistors to provide for the proper voltage and current requirements of the pulse sequence.

8.7.1 DC MOTOR OPERATING PARAMETERS

Space does not allow a full discussion of all motor types. We will concentrate on the DC motor. As previously mentioned, the motor speed may be varied by changing the applied voltage. This is difficult to do with a digital control signal. However, PWM control signal techniques discussed earlier may be combined with a MOSFET interface to precisely control the motor speed. The duty cycle of the PWM signal will also be the percentage of the motor supply voltage applied to the motor, and hence the percentage of rated full speed at which the motor will rotate. The interface circuit to accomplish this type of control is shown in Figure 8.16. Various portions of this interface circuit have been previously discussed. The resistor R_G, typically 10 k ohm, is used to discharge the MOSFET gate when no voltage is applied to the gate. For an inductive load, a reversed biased protection diode must be across the load. The interface circuit shown allows the motor to rotate in a given direction. As previously mentioned, to rotate the motor in the opposite direction, the motor polarity must be reversed. This may be accomplished with a high power switching network called an H-bridge specifically designed for this purpose. Reference Pack and Barrett for more information on this topic.

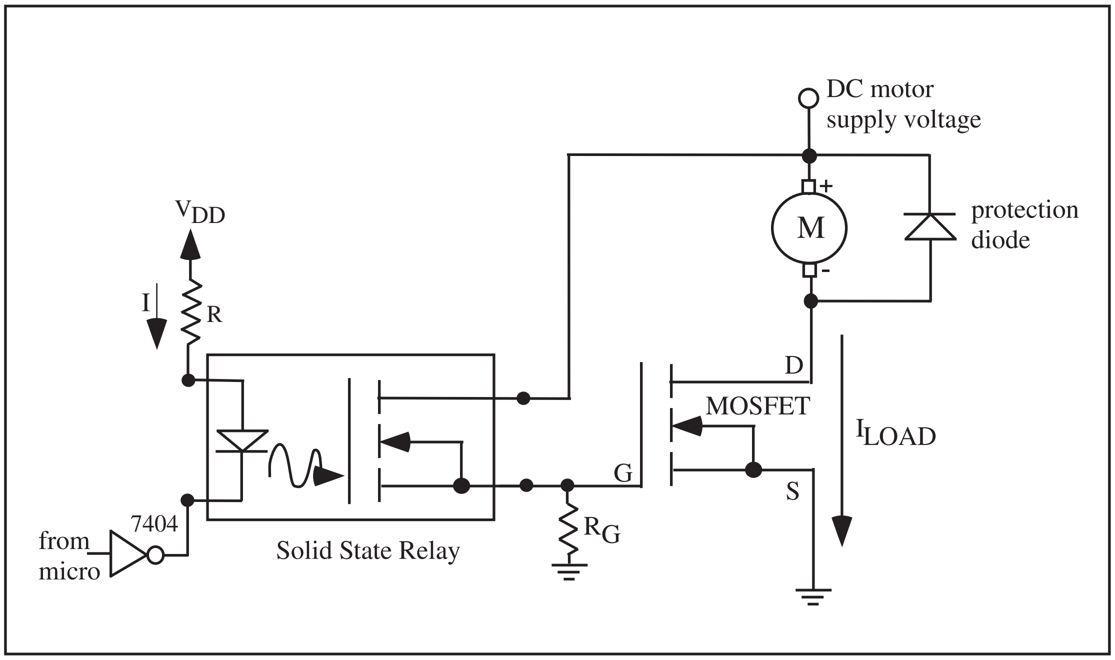


Figure 8.16: DC motor interface.

8.7.2 H-BRIDGE DIRECTION CONTROL

For a DC motor to operate in both the clockwise and counter clockwise direction, the polarity of the DC motor supplied must be changed. To operate the motor in the forward direction, the positive battery terminal must be connected to the positive motor terminal while the negative battery terminal must be provided to the negative motor terminal. To reverse the motor direction the motor supply polarity must be reversed. An H-bridge is a circuit employed to perform this polarity switch. Low power H-bridges (500 mA) come in a convenient dual in line package (e.g., 754110). For higher power motors, a H-bridge may be constructed from discrete components as shown in Figure 8.17. If PWM signals are used to drive the base of the transistors (from microcontroller pins PD4 and PD5), both motor speed and direction may be controlled by the circuit. The transistors used in the circuit must have a current rating sufficient to handle the current requirements of the motor during start and stall conditions.

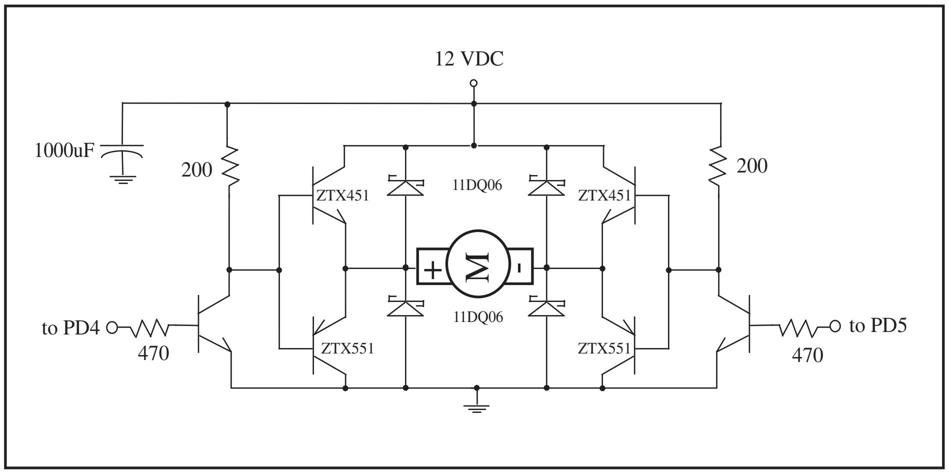


Figure 8.17: H-bridge control circuit.

8.7.3 SERVO MOTOR INTERFACE

The servo motor is used for a precise angular displacement. The displacement is related to the duty cycle of the applied control signal. A servo control circuit and supporting software was provided in Chapter 7.

8.7.4 STEPPER MOTOR CONTROL

Stepper motors are used to provide a discrete angular displacement in response to a control signal step. There are a wide variety of stepper motors including bipolar and unipolar types with different configurations of motor coil wiring. Due to space limitations, we only discuss the unipolar, 5 wire stepper motor. The internal coil configuration for this motor is provided in Figure 8.18(b).

Often, a wiring diagram is not available for the stepper motor. Based on the wiring configuration (Reference Figure 8.18b), one can find out the common line for both coils. It will have a resistance that is one-half of all of the other coils. Once the common connection is found, one can connect the stepper motor into the interface circuit. By changing the other connections, one can determine the correct connections for the step sequence.

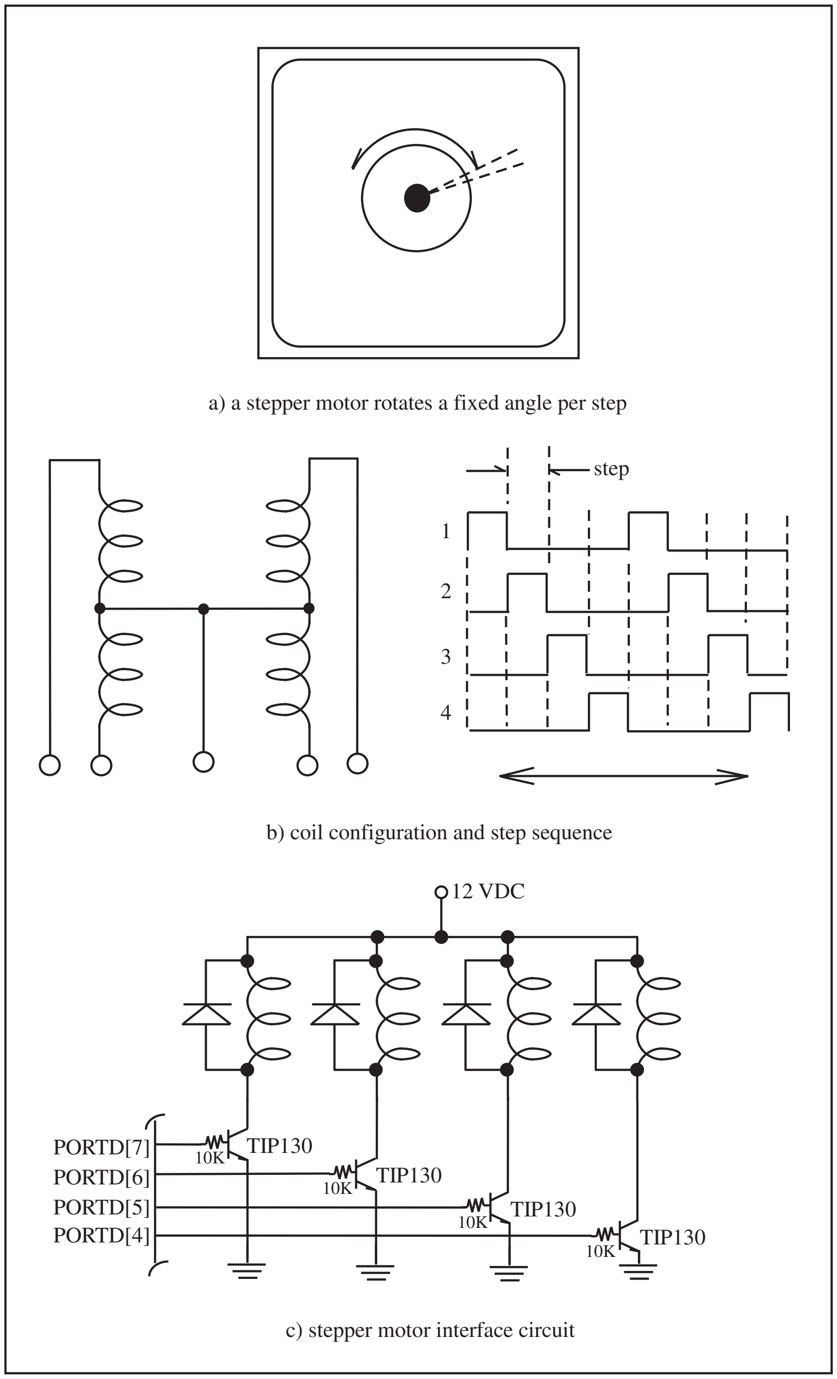

a) a stepper motor rotates a fixed angle per step
step
1
2
3
4
b) coil configuration and step sequence
12 VDC
PORTD[7]
PORTD[6]
PORTD[5]
PORTD[4]
10K
TIP130
c) stepper motor interface circuit

Figure 8.18: Unipolar stepper motor control circuit.

To rotate the motor either clockwise or counter clockwise, a specific step sequence must be sent to the motor control wires as shown in Figure 8.18(c). As shown in Figure 8.18(c), the control sequence is transmitted by four pins on the microcontroller. In this example, we use PORTD[7:5].

The microcontroller does not have sufficient capability to drive the motor directly. Therefore, an interface circuit is required as shown in Figure 8.18c). For a unipolar stepper motor, we employ a TIP130 power Darlington transistor to drive each coil of the stepper motor. The speed of motor rotation is determined by how fast the control sequence is completed. The TIP 30 must be powered by a supply that has sufficient capability for the stepper motor coils.

Example: An ATmega324 has been connected to a JRP 42BYG016 unipolar, 1.8 degree per step, 12 VDC at 160 mA stepper motor. The interface circuit is shown in Figure 8.19. PORTD pins 7 to 4 are used to provide the step sequence. A one second delay is used between the steps to control motor speed. Pushbutton switches are used on PORTB[1:0] to assert CW and CCW stepper motion. An interface circuit consisting of four TIP130 transistors are used between the microcontroller and the stepper motor to boost the voltage and current of the control signals. Code to provide the step sequence is shown below.

Provided below is a basic function to rotate the stepper motor in the forward or reverse direction.

```
//*************************************************************************
//target controller: ATMEL ATmega328
//
//ATMEL AVR ATmega328PV Controller Pin Assignments
//Pin  1 Reset - 1M resistor to Vdd, tact switch to ground, 1.0 uF to ground
//Pin  6 PD4 - to stepper motor coil
//Pin  7 Vdd - 1.0 uF to ground
//Pin  8 Gnd
//Pin 11 PD5 - to stepper motor coil
//Pin 12 PD6 - to stepper motor coil
//Pin 13 PD7 - to stepper motor coil
//Pin 14 PB0 to active high RC debounced switch - CW
//Pin 20 AVcc to Vdd
//Pin 21 ARef to Vcc
//Pin 22 Gnd to Ground
//*************************************************************************

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

//ATMEL register definitions for ATmega328
#include<iom328pv.h>
#include<macros.h>
```

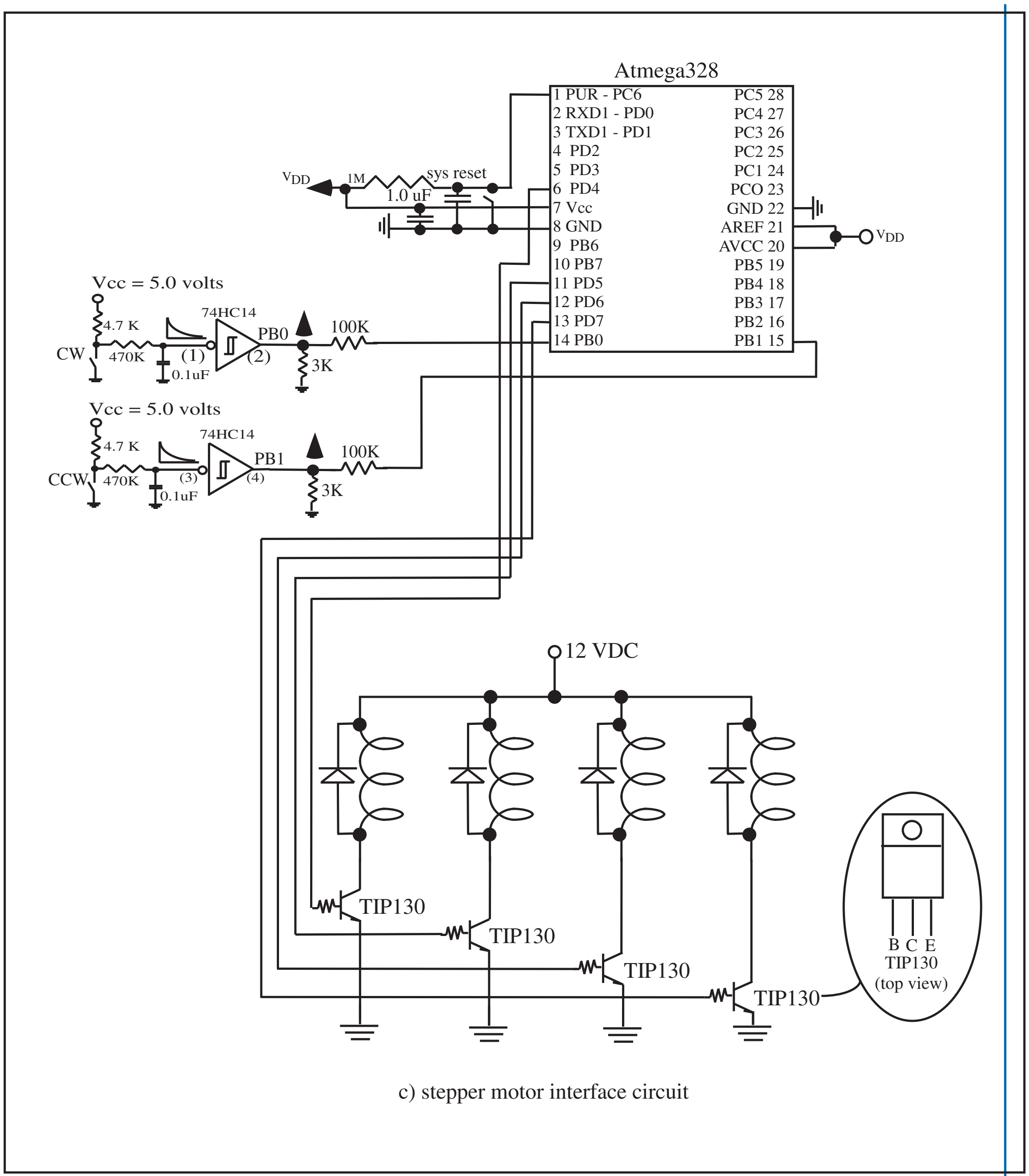


Figure 8.19: Unipolar stepper motor control circuit.

```
                                      //interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:17


//function prototypes*****************************************************
void initialize_ports(void);          //initializes ports
void read_new_input(void);            //used to read input change on PORTB
void init_timer0_ovf_interrupt(void); //used to initialize timer0 overflow
void timer0_interrupt_isr(void);
void delay(unsigned int);


//main program************************************************************
//The main program checks PORTB for user input activity. If new activity
//is found, the program responds.

//global variables
unsigned char   old_PORTB = 0x08; //present value of PORTB
unsigned char   new_PORTB;        //new values of PORTB
unsigned int    input_delay;      //delay counter - increment via Timer0
                                  //overflow interrupt

void main(void)
{
initialize_ports();               //return LED configuration to default
init_timer0_ovf_interrupt();      //used to initialize timer0 overflow

while(1)
  {
  _StackCheck();                  //check for stack overflow
  read_new_input();               //read input status changes on PORTB
  }
}//end main

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

```
void initialize_ports(void)
{
//PORTB
DDRB=0xfc;                          //PORTB[7-2] output, PORTB[1:0] input
PORTB=0x00;                         //disable PORTB pull-up resistors

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

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

//***************************************************************************
//read_new_input: functions polls PORTB for a change in status. If status
//change has occurred, appropriate function for status change is called
//Pin 1 PB0 to active high RC  debounced switch - CW
//Pin 2 PB1 to active high RC debounced switch  - CCW
//***************************************************************************

void read_new_input(void)
{
new_PORTB = (PINB & 0x03);
if(new_PORTB != old_PORTB){
  switch(new_PORTB){                //process change in PORTB input

    case 0x01:                      //CW
      while((PINB & 0x03)==0x01)
        {
        PORTD = 0x80;
        delay(15);                  //delay 1s
        PORTD = 0x00;
        delay(1);                   //delay 65 ms

        PORTD = 0x40;
        delay(15);
        PORTD = 0x00;
```

```
          delay(1);

          PORTD = 0x20;
          delay(15);
          PORTD = 0x00;
          delay(1);

          PORTD = 0x10;
          delay(15);
          PORTD = 0x00;
          delay(1);
          }
   break;

    case 0x02:                     //CCW
      while((PINB & 0x03)==0x02)
        {
        PORTD = 0x10;
        delay(15);
        PORTD = 0x00;
        delay(1);

        PORTD = 0x20;
        delay(15);
        PORTD = 0x00;
        delay(1);

        PORTD = 0x40;
        delay(15);
        PORTD = 0x00;
        delay(1);

        PORTD = 0x80;
        delay(15);
        PORTD = 0x00;
        delay(1);
  }
        break;
       default:;                   //all other cases
       }                           //end switch(new_PORTB)
   }                               //end if new_PORTB
   old_PORTB=new_PORTB;            //update PORTB
 }

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

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

 //*************************************************************************
 //timer0_interrupt_isr:
 //Note: Timer overflow 0 is cleared by hardware when executing the
 //corresponding interrupt handling vector.
 //*************************************************************************

 void timer0_interrupt_isr(void)
 {
 input_delay++;                    //input delay processing
 }

 //*************************************************************************
 //void delay(unsigned int number_of_65_5ms_interrupts)
 //this generic delay function provides the specified delay as the number
 //of 65.5 ms "clock ticks" from the Timer0 interrupt.
 //Note: this function is only valid when using a 1 MHz crystal or ceramic
 //      resonator
```

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

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

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

8.7.5 AC DEVICES

In a similar manner, a high power alternating current (AC) load may be switched on and off using a low power control signal from the microcontroller. In this case, a Solid State Relay is used as the switching device. Solid state relays are available to switch a high power DC or AC load [Crydom]. For example, the Crydom 558-CX240D5R is a printed circuit board mounted, air cooled, single pole single throw (SPST), normally open (NO) solid state relay. It requires a DC control voltage of 3-15 VDC at 15 mA. However, this small microcontroller compatible DC control signal is used to switch 12-280 VAC loads rated from 0.06 to 5 amps [Crydom] as shown in Figure 8.20.

To vary the direction of an AC motor you must use a bi-directional AC motor. A bi-directional motor is equipped with three terminals: common, clockwise, and counterclockwise. To turn the motor clockwise, an AC source is applied to the common and clockwise connections. In like manner, to turn the motor counterclockwise, an AC source is applied to the common and counterclockwise connections. This may be accomplished using two of the Crydom SSRs.

8.8 INTERFACING TO MISCELLANEOUS DEVICES

In this section, we provide a pot pourri of interface circuits to connect a microcontroller to a wide variety of peripheral devices.

8.8.1 SONALERTS, BEEPERS, BUZZERS

In Figure 8.21, we provide several circuits used to interface a microcontroller to a buzzer, beeper or other types of annunciator indexannunciator devices such as a sonalert. It is important that the interface transistor and the supply voltage are matched to the requirements of the sound producing device.

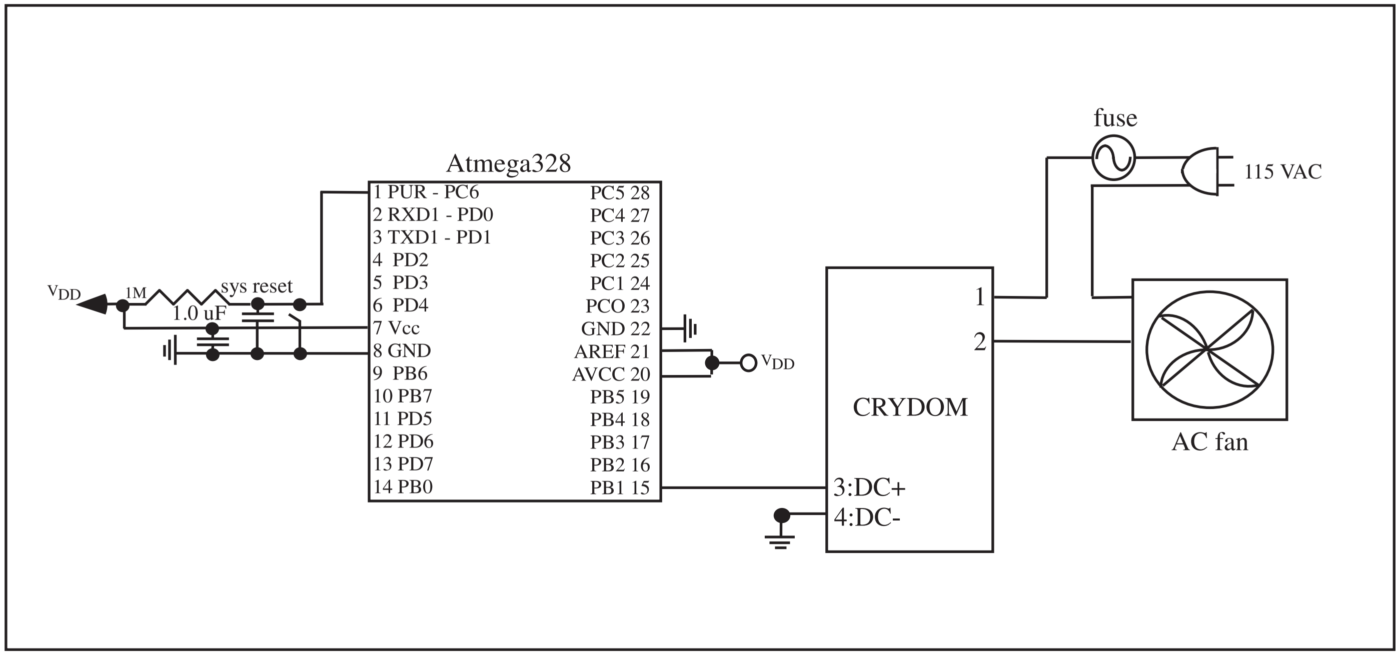


Figure 8.20: AC motor control circuit.

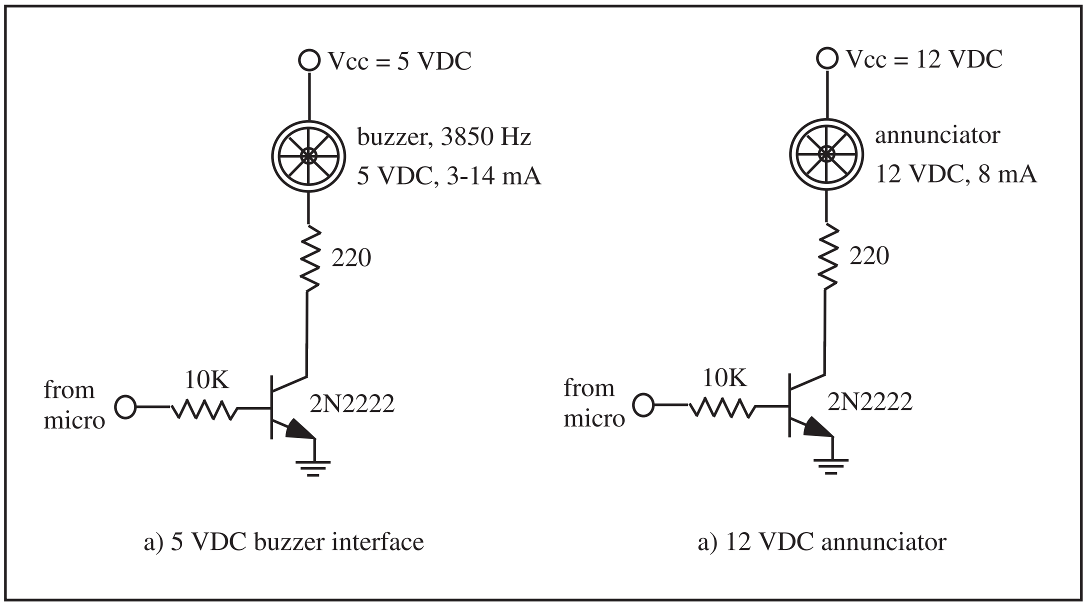


Figure 8.21: Sonalert, beepers, buzzers.

8.8.2 VIBRATING MOTOR

A vibrating motor is often used to gain one's attention as in a cell phone. These motors are typically rated at 3 VDC and a high current. The interface circuit shown in Figure 8.22 is used to drive the low voltage motor. Note that the control signal provided to the transistor base is 5 VDC. To step the motor supply voltage down to the motor voltage of 3 VDC, two 1N4001 silicon rectifier diodes are used in series. These diodes provide approximately 1.4 to 1.6 VDC voltage drop. Another 1N4001 diode is reversed biased across the motor and the series diode string. The motor may be turned on and off with a 5 VDC control signal from the microcontroller or a PWM signal may be used to control motor speed.

8.9 EXTENDED EXAMPLE 1: AUTOMATED FAN COOLING SYSTEM

In this section, we describe an embedded system application to control the temperature of a room or some device. The system is illustrated in Figure 8.23. An LM34 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).

We send a 1 KHz PWM signal to a cooling fan (M) whose duty cycle is set from 50% to 90% using the potentiometer connected to PORTC[2]. The PWM signal should last until the

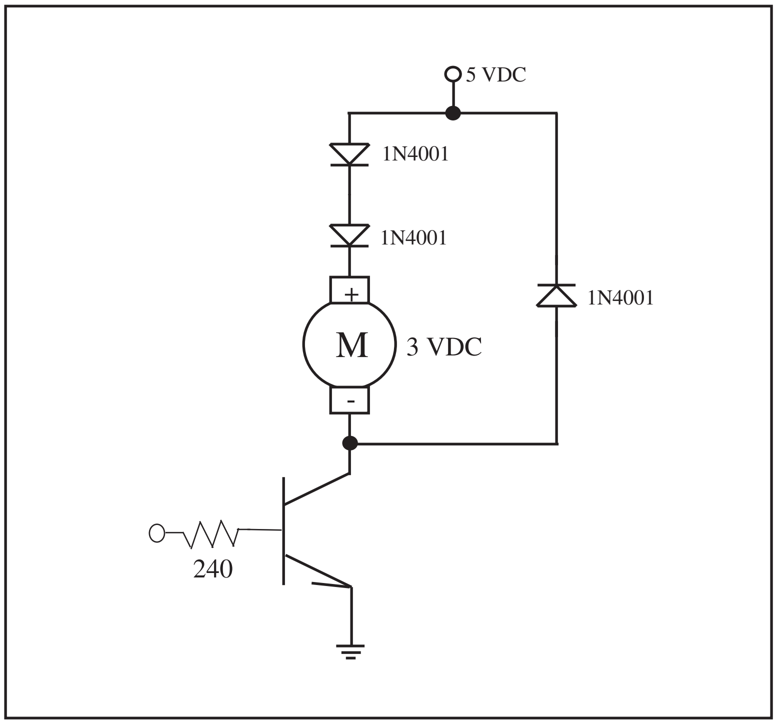


Figure 8.22: Controlling a low voltage motor.

temperature of the LM34 cools to a value as set by another potentiometer (PORTC[1]). When the temperature of the LM34 falls below the 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.

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 10, 2010
//*************************************************************************
//Description: This program reads the voltage from an LM34 temperature
//sensor then sends the corresponding temperature to an LCD.
//If the sensed temperature is greater than the temperature designated
```

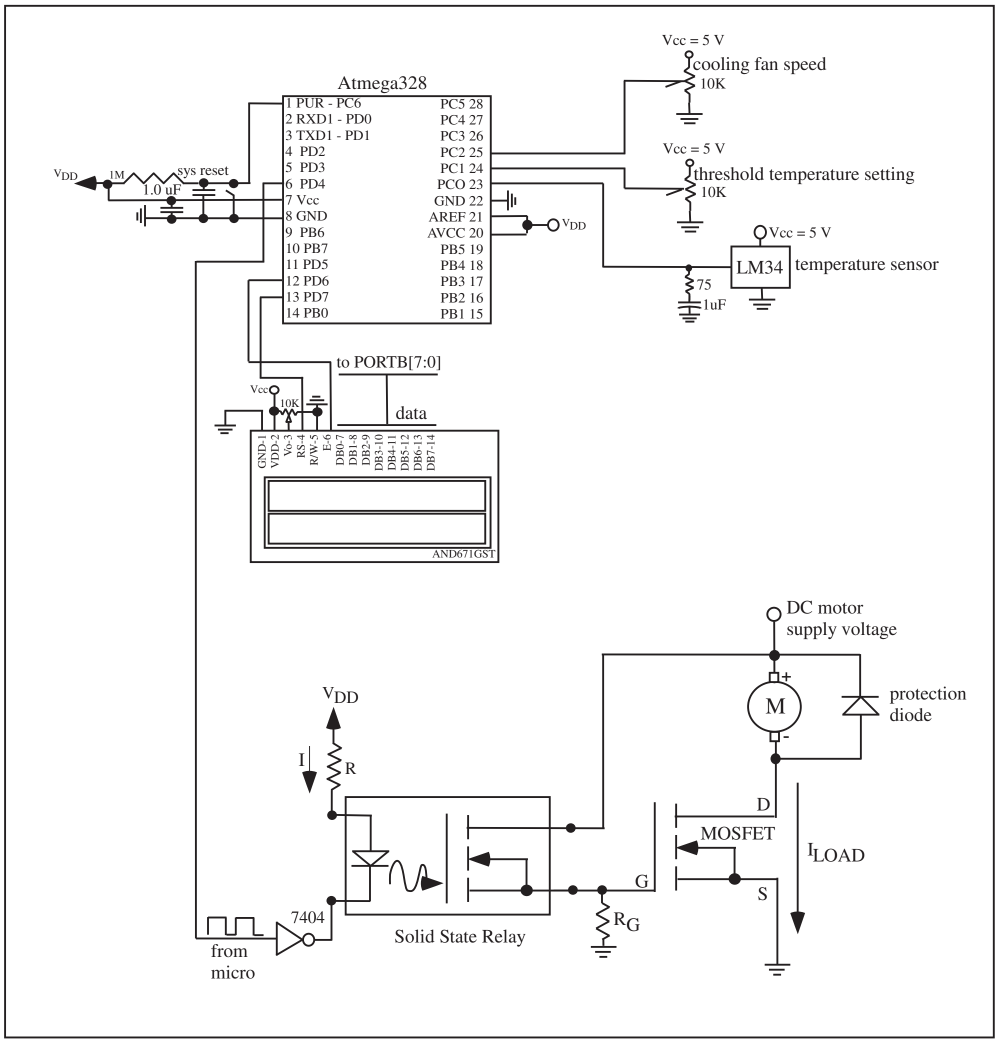


Figure 8.23: Automated fan cooling system.

```
//by a potentiometer, then a PWM signal is turned on to trigger a fan
//with duty cycle designated by another potentiometer.
//
//Ports:
//  PORTB[7:0]: data output to LCD
//  PORTD[7:6]: LCD control pins
//  PORTC[2:0]:
//  PORTC[0]: LM34 temperature sensor
//  PORTC[1]: threshold temperature
//  PORTC[2]: fan speed
//  PORTD[4]  : PWM channel B output
//
//***************************************************************************

//include files*************************************************************
#include<iom328pv.h>

//function prototypes*******************************************************
void initializePorts();
void initializeADC();
unsigned int readADC(unsigned char);
void LCD_init();
void putChar(unsigned char);
void putcommand(unsigned char);
void voltageToLCD(unsigned int);
void temperatureToLCD(unsigned int);
void PWM(unsigned int);
void delay_5ms();

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()
{
DDRD = 0xFF;
DDRC = 0xFF;
DDRB = 0xFF;
}

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

void initializeADC()
{
//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

                                       //Set ADC module prescalar to 8
                                       //critical accurate ADC results
while (!(ADCSRA & 0x10));              //Check if conversion is ready
ADCSRA |= 0x10;                        //Clear conv rdy flag - set the bit

binary_weighted_voltage_low = ADCL;
 //Read 8 low bits first (important)
   //Read 2 high bits, multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage = binary_weighted_voltage_low +
                          binary_weighted_voltage_high;

return binary_weighted_voltage;        //ADCH:ADCL
}

//*************************************************************************
//LCD_Init: initialization for an LCD connected in the following manner:
//LCD: AND671GST 1x16 character display
//LCD configured as two 8 character lines in a 1x16 array
//LCD data bus (pin 14-pin7) ATMEL ATmega16: PORTB
//LCD RS (pin 4) ATMEL ATmega16: PORTD[7]
//LCD E (pin 6) ATMEL ATmega16: PORTD[6]
```

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


void LCD_init(void)
{
delay_5ms();
delay_5ms();
delay_5ms();
                                       // output command string to
                                       //initialize LCD
putcommand(0x38);                      //function set 8-bit
delay_5ms();
putcommand(0x38);                      //function set 8-bit
delay_5ms();
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_5ms();
}
```

```
//*************************************************************************
//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_5ms();
}


//*************************************************************************
//delays for 5 ms with a clock speed of 1 MHz
//*************************************************************************

void delay_5ms(void)
{
unsigned int i;

for(i=0; i<2500; i++)
  {
  asm("nop");
  }
}

//*************************************************************************
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% = 127,
//  5 VDC = 100% = 255

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);//set PWM duty cycle Ch B to 0%
                                        //Ramp up 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);
  }
}

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

8.10 EXTENDED EXAMPLE 2: FINE ART LIGHTING SYSTEM

In Chapter 2 and 5, we investigated an illumination system for a painting. The painting was to be illuminated via high intensity white LEDs. The LEDs could be mounted in front of or behind the painting as the artist desired. In Chapter 5, we equipped the lighting system with an IR sensor to detect the presence of someone viewing the piece. We also wanted to adjust the intensity of the lighting based on how the close viewer was to the art piece. In this example, we enhance the lighting system with three IR sensors. The sensors will determine the location of viewers and illuminate portions of the painting accordingly. The analogWrite function is used to set the intensity of the LEDs using PWM techniques.

The Arduino Development Environment sketch to sense how away the viewer is and issue a proportional intensity control signal to illuminate the LED is provided below. The analogRead function is used to obtain the signal from the IR sensor. The analogWrite function is used to issue a proportional signal.

```
//*************************************************************************
                                     //analog input pins
#define left_IR_sensor   0           //analog pin - left IR sensor
#define center_IR_sensor 1           //analog pin - center IR sensor
#define right_IR_sensor  2           //analog pin - right IR sensor

                                     //digital output pins
                                     //LED indicators - wall detectors
#define col_0_control      0         //digital pin - column 0 control
#define col_1_control      1         //digital pin - column 1 control
#define col_2_control      2         //digital pin - column 2 control

int left_IR_sensor_value;            //declare var. for left IR sensor
int center_IR_sensor_value;          //declare var. for center IR sensor

int right_IR_sensor_value;           //declare var. for right IR sensor

void setup()
  {
                                     //LED indicators - wall detectors
  pinMode(col_0_control, OUTPUT);    //configure pin 0 for digital output
  pinMode(col_1_control, OUTPUT);    //configure pin 1 for digital output
```

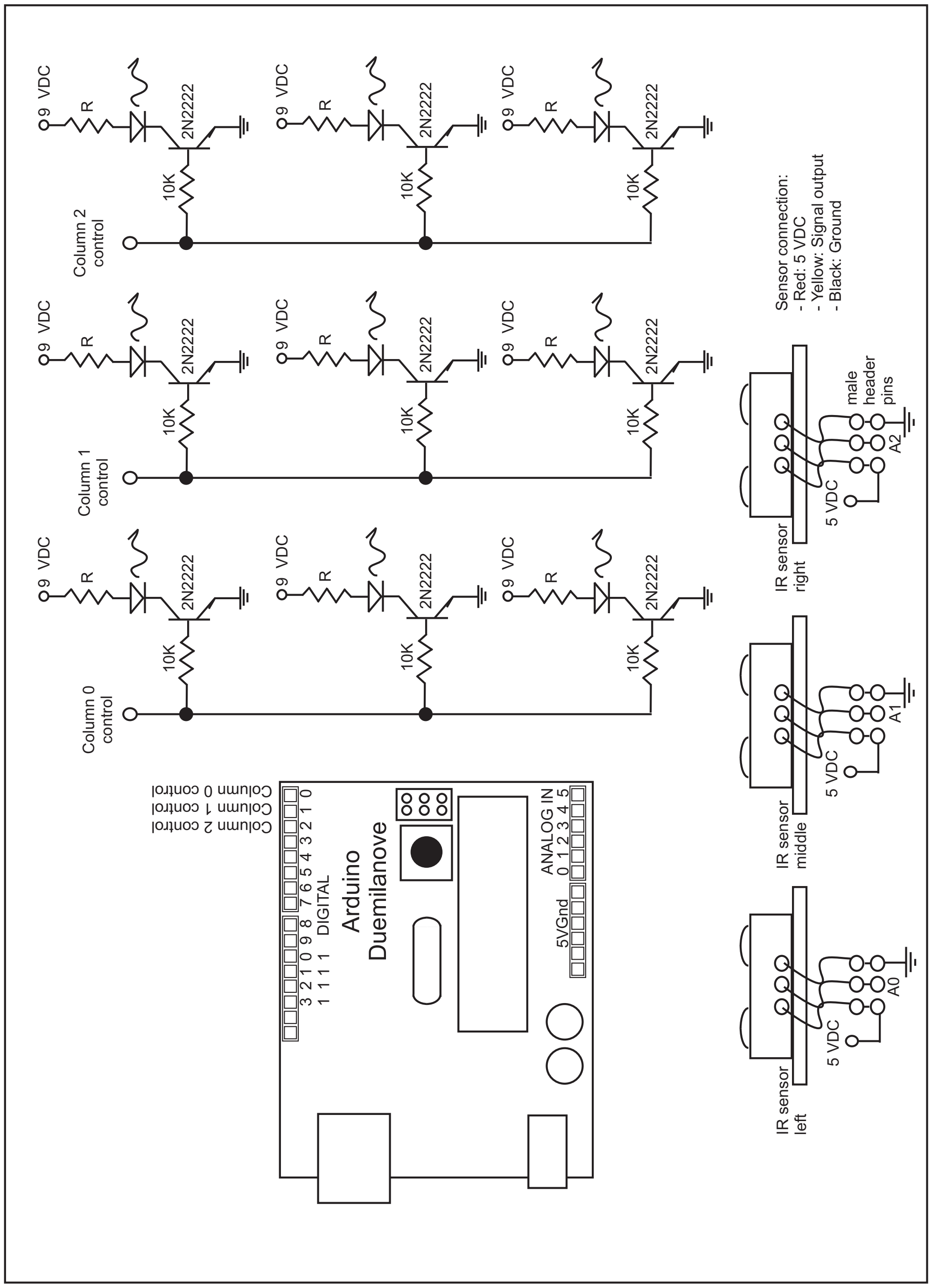


Figure 8.24: IR sensor interface.

```
  pinMode(col_2_control, OUTPUT);    //configure pin 2 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);

  if(left_IR_sensor_value < 128)
    {
    analogWrite(col_0_control, 31);   //0 (off) to 255 (full speed)
    }
  else if(left_IR_sensor_value < 256)
    {
    analogWrite(col_0_control, 63);   //0 (off) to 255 (full speed)
    }
  else if(left_IR_sensor_value < 384)
    {
    analogWrite(col_0_control, 95);   //0 (off) to 255 (full speed)
    }
  else if(left_IR_sensor_value < 512)
    {
    analogWrite(col_0_control, 127);  //0 (off) to 255 (full speed)
    }
  else if(left_IR_sensor_value < 640)
    {
    analogWrite(col_0_control, 159);  //0 (off) to 255 (full speed)
    }
  else if(left_IR_sensor_value < 768)
    {
    analogWrite(col_0_control, 191);  //0 (off) to 255 (full speed)
    }
  else if(left_IR_sensor_value < 896)
    {
    analogWrite(col_0_control, 223);  //0 (off) to 255 (full speed)
    }
  else
    {
    analogWrite(col_0_control, 255);  //0 (off) to 255 (full speed)
    }

if(center_IR_sensor_value < 128)
     {
     analogWrite(col_1_control, 31);  //0 (off) to 255 (full speed)
     }
   else if(center_IR_sensor_value < 256)
     {
     analogWrite(col_1_control, 63);  //0 (off) to 255 (full speed)
     }
   else if(center_IR_sensor_value < 384)
     {
     analogWrite(col_1_control, 95);  //0 (off) to 255 (full speed)
     }
   else if(center_IR_sensor_value < 512)
     {
     analogWrite(col_1_control, 127); //0 (off) to 255 (full speed)
     }
   else if(center_IR_sensor_value < 640)
     {
     analogWrite(col_1_control, 159); //0 (off) to 255 (full speed)
     }
   else if(center_IR_sensor_value < 768)
     {
     analogWrite(col_1_control, 191); //0 (off) to 255 (full speed)
     }
   else if(center_IR_sensor_value < 896)
     {
     analogWrite(col_1_control, 223); //0 (off) to 255 (full speed)
     }
   else
     {
     analogWrite(col_1_control, 255); //0 (off) to 255 (full speed)
     }

if(right_IR_sensor_value < 128)
```

```
     {
     analogWrite(col_2_control, 31);  //0 (off) to 255 (full speed)
     }
   else if(right_IR_sensor_value < 256)
     {
     analogWrite(col_2_control, 63);  //0 (off) to 255 (full speed)
     }
   else if(right_IR_sensor_value < 384)
     {
     analogWrite(col_2_control, 95);  //0 (off) to 255 (full speed)
     }
   else if(right_IR_sensor_value < 512)
     {
     analogWrite(col_2_control, 127); //0 (off) to 255 (full speed)
     }
   else if(right_IR_sensor_value < 640)
     {
     analogWrite(col_2_control, 159); //0 (off) to 255 (full speed)
     }
  else if(right_IR_sensor_value < 768)
    {
    analogWrite(col_2_control, 191);  //0 (off) to 255 (full speed)
    }
  else if(right_IR_sensor_value < 896)
    {
    analogWrite(col_2_control, 223);  //0 (off) to 255 (full speed)
    }
  else
    {
    analogWrite(col_2_control, 255);  //0 (off) to 255 (full speed)
    }

delay(500);                           //delay 500 ms
}

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

8.11 EXTENDED EXAMPLE 3: FLIGHT SIMULATOR PANEL

We close the chapter with an extended example of developing a flight simulator panel. This panel was actually designed and fabricated for a middle school to allow students to react to space mission like status while using a program that allowed them to travel about the planets. An Atmel ATmega328 microcontroller is used because its capabilities best fit the requirements for the project.

The panel face is shown in Figure 8.25. It consists of a joystick that is connected to a host computer for the flight simulator software. Below the joystick is a two line liquid crystal display equipped with a backlight LED (Hantronix HDM16216L-7, Jameco# 658988). Below the LCD is a buzzer to alert students to changing status. There are also a complement of other status indicators. From left-to-right is the Trip Duration potentiometer. At the beginning of the flight episode students are prompted to set this from 0 to 60 minutes to communicate to the microcontroller the length of the flight episode. This data is used to calculate different flight increments. There are also a series of simulated circuit breakers: system reset (SYS Reset), oxygen (O_2 CB), auxiliary fuel (AUX FUEL CB), and the main power circuit breakers (MAIN PWR CB). These are not actual circuit breakers but normally open (NO) single pole single throw (SPST) momentary pushbutton switches that allow the students to interact with the microcontroller. There are also a series of LEDs that form a Y pattern on the panel face. They are also used to indicate status changes.

To interface the flight simulator panel to the microcontroller a number of different techniques previously discussed in the book were employed. The interface diagram is shown in Figure 8.26. Pin 1 is a reset for the microcontroller. When the switch is depressed, pin 1 is taken low and resets the microcontroller. Port D of the microcontroller (pins 2-6, 11-13) forms the data connection for the LCD. Pin 9 is used to turn the buzzer on and off. This pin is routed through a transistor interface described earlier in the text. Port B[5:0] (pins 14-19) is used to control the LEDs on the front panel. Each individual LED is also supported with a transistor interface circuit. Conveniently, these small NPN signal transistors come in four to a 14 pin DIP package (MPQ2222). Port C [0] (pin 23) is used as an analog input pin. It is connected to the trip duration potentiometer. Port C pins [3:1] (pins 24 to 26) are used to connect the NO SPST tact switches to the microcontroller. Port C pins [4:5] (pins 27, 28) are used for the LCD control signals.

The software flowchart is shown in Figure 8.27. After startup the students are prompted via the LCD display to set the trip duration and then press the main power circuit breaker. This potentiometer setting is then used to calculate four different trip increments. Countdown followed by blastoff then commences. At four different trip time increments, students are presented with status that they must respond to. Real clock time is kept using the TCNT0 timer overflow configured as a 65.5 ms "clock tick." The overall time base for the microcontroller was its internal 1 MHz clock that may be selected during programming with the STK500.

Provided below is the code listing for the flight simulator panel.

```
//*********************************************************************
//file name: flight_sim.c
//author: Steve Barrett, Ph.D., P.E.
```

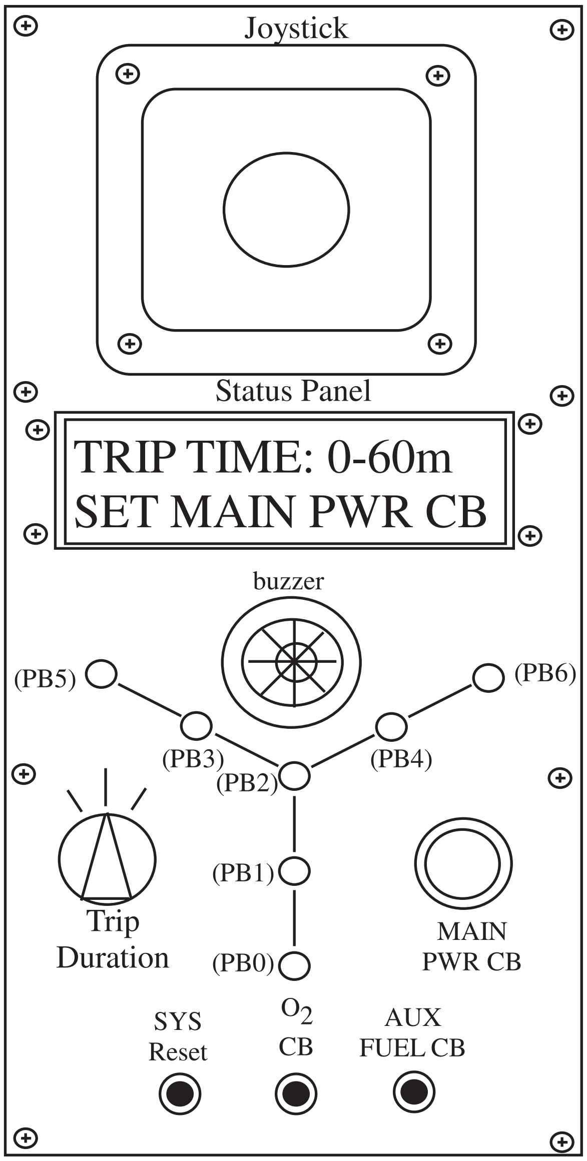


Figure 8.25: Flight Simulator Panel.

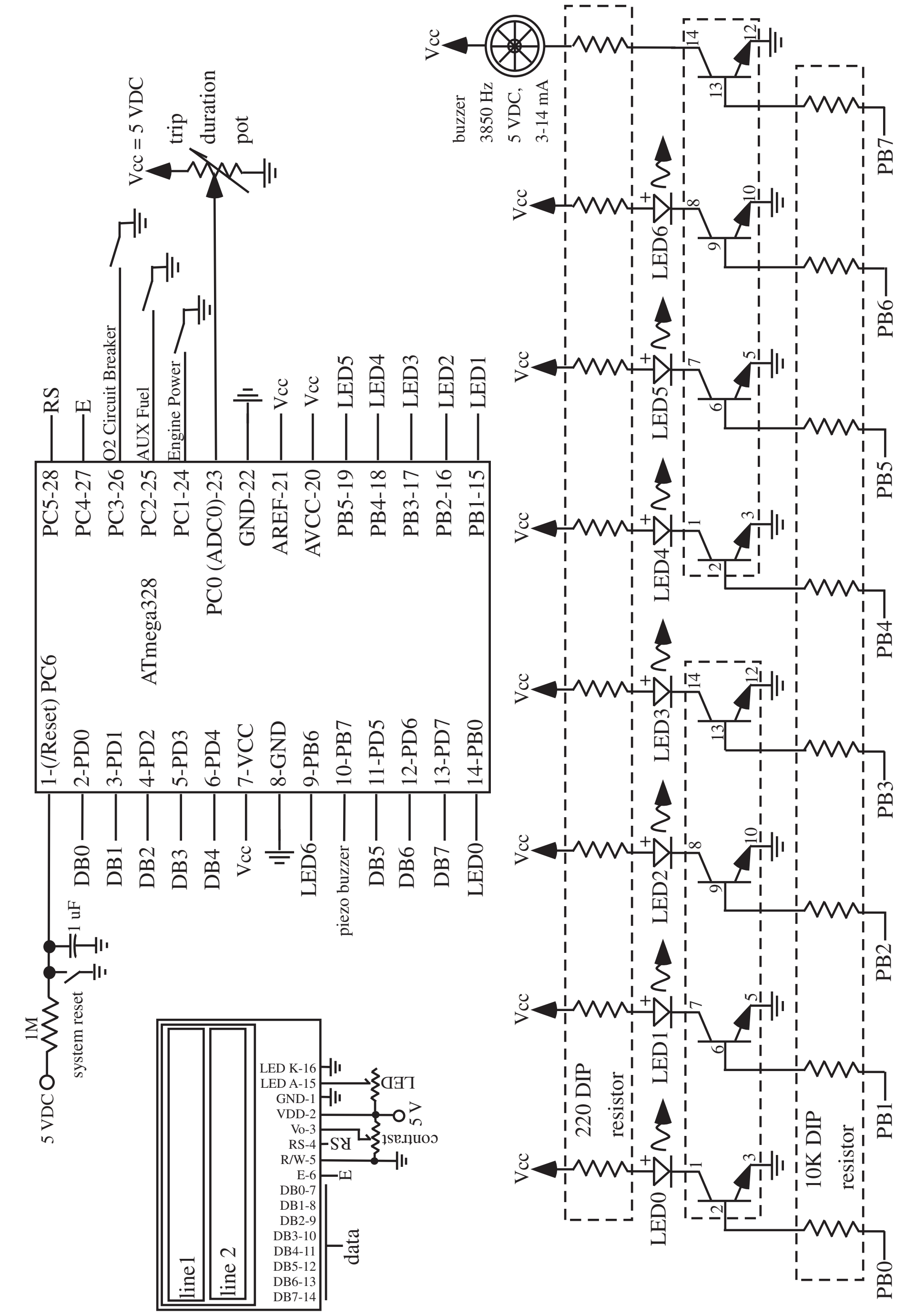


Figure 8.26: Interface diagram for the flight simulator panel.

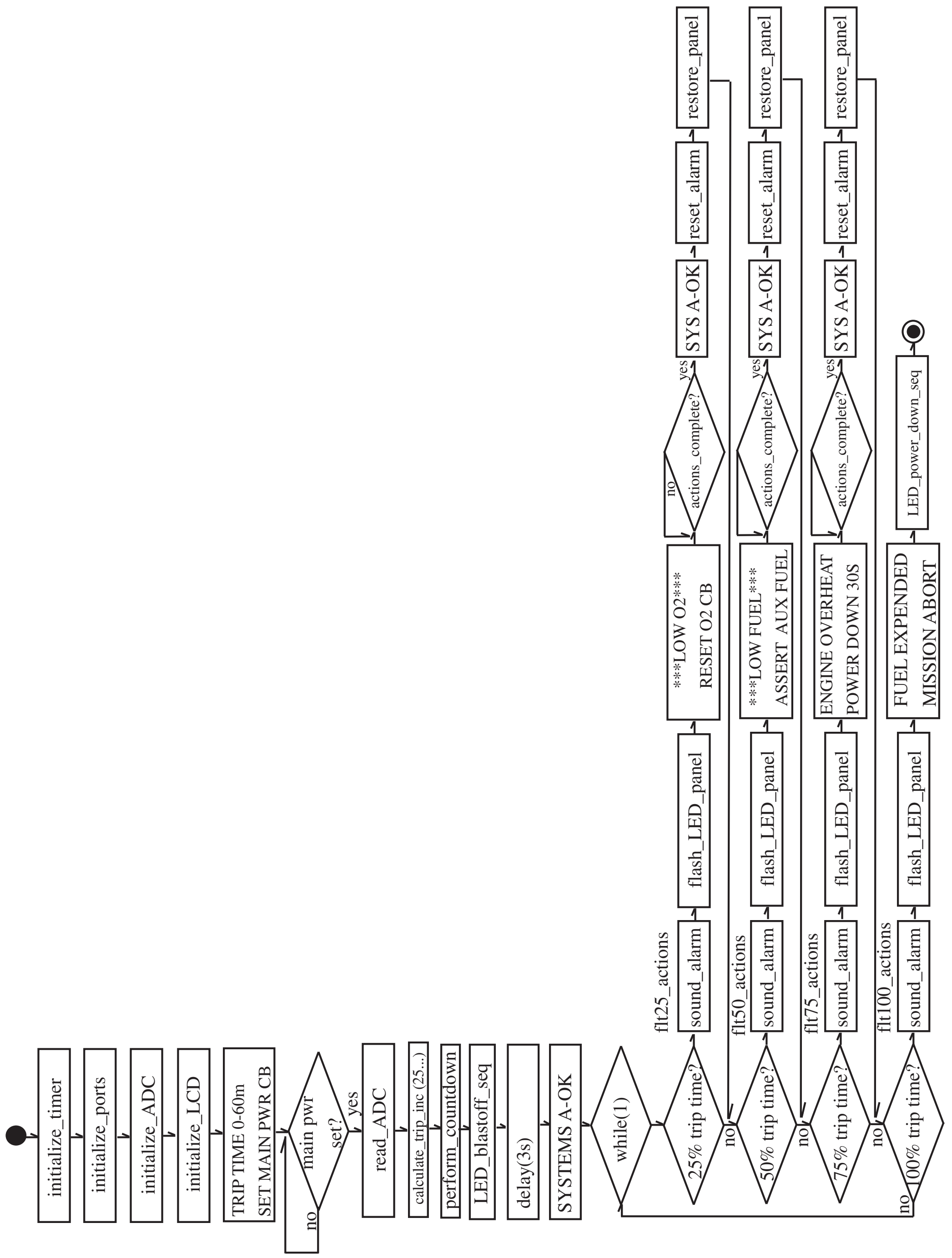


Figure 8.27: Software flow for the flight simulator panel.

```
//last revised: April 10, 2010
//function: Controls Flight Simulator Control Panel for Larimer County
//          School District #1
//
//ATMEL AVR ATmega328
//Chip Port Function I/O Source/Dest Asserted Notes
//***************************************************************************
//Pin 1: /Reset
//Pin 2: PD0 to DB0 LCD
//Pin 3: PD1 to DB1 LCD
//Pin 4: PD2 to DB2 LCD
//Pin 5: PD3 to DB3 LCD
//Pin 6: PD4 to DB4 LCD
//Pin 7: Vcc
//Pin 8: Gnd
//Pin 9: PB6 to LED6
//Pin 10: PB7 to piezo buzzer
//Pin 11: PD5 to DB6 LCD
//Pin 12: PD6 to DB6 LCD
//Pin 13: PD7 to DB7 LCD
//Pin 14: PB0 to LED0
//Pin 15: PB1 to LED1
//Pin 16: PB2 to LED2
//Pin 17: PB3 to LED3
//Pin 18: PB4 to LED4
//Pin 19: PB5 to LED5
//Pin 20: AVCC to Vcc
//Pin 21: AREF to Vcc
//Pin 22 Gnd
//Pin 23 ADC0 to trip duration potentiometer
//Pin 24 PC1 Engine Power Switch
//Pin 25 PC2 AUX Fuel circuit breaker
//Pin 26 PC3 O2 circuit breaker
//Pin 27 PC4 to LCD Enable (E)
//Pin 28 PC5 to LCD RS
//

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

```
//ATMEL register definitions for ATmega8
#include<iom328v.h>



//function prototypes****************************************************
void delay(unsigned int number_of_65_5ms_interrupts);
void init_timer0_ovf_interrupt(void);
void InitADC(void);                          //initialize ADC
void initialize_ports(void);                 //initializes ports
void power_on_reset(void);
    //returns system to startup state
unsigned int  ReadADC(unsigned char chan);//read value from ADC results
void clear_LCD(void);                        //clears LCD display
void LCD_Init(void);                         //initialize AND671GST LCD
void putchar(unsigned char c);               //send character to LCD
void putcommand(unsigned char c);            //send command to LCD
unsigned int  ReadADC(unsigned char chan);//read value from ADC results
void timer0_interrupt_isr(void);
void flt25_actions(void);
void flt50_actions(void);
void flt75_actions(void);
void flt100_actions(void);
void sound_alarm(void);
void turn_off_LEDs(void);
void reset_alarm(void);
void restore_panel(void);
void LED_blastoff_sequence(void);
void LED_power_down_sequence(void);
void monitor_main_power_CB(void);
void monitor_O2_CB_reset(void);
void monitor_aux_fuel_CB(void);
void perform_countdown(void);
void print_LOWO2(void);
void print_LOW_FUEL(void);
void print_fuel_expended(void);
void print_OVERHEAT(void);
void print_trip_dur(void);
void flash_LED_panel(void);
void clear_LCD(void);
void calculate_trip_int(void);
void systems_A_OK(void);

//program constants
#define TRUE     1
#define FALSE     0
#define OPEN     1
#define CLOSE     0
#define YES     1
#define NO     0
#define SAFE     1
#define UNSAFE     0
#define ON          1
#define OFF         0

//interrupt handler definition
#pragma interrupt_handler timer0_interrupt_isr:17

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

//global variables
unsigned int flt_25, flt_50, flt_75, flt_100;
unsigned int action25_done=NO, action50_done=NO;
unsigned int action75_done=NO, action100_done=NO;
unsigned int achieved25=NO, achieved50=NO;
unsigned int achieved75=NO, achieved100=NO;
unsigned int flt_timer=0;
unsigned int trip_duration_volt;
unsigned char PORTC_pullup_mask = 0x0e;
unsigned int flash_timer;
unsigned int PORTB_LEDs;
unsigned int flash_panel=NO;
unsigned int delay_timer;
unsigned int troubleshooting = 1;
void convert_display_voltage_LCD(int trip_duration_volt);
void convert_int_to_string_display_LCD(unsigned int total_integer_value);

void main(void)
{
```

```
init_timer0_ovf_interrupt();
//initialize Timer0 to serve as elapsed
initialize_ports();                    //initialize ports
InitADC();                             //initialize ADC
LCD_Init();                            //initialize LCD
print_trip_dur();
//prompt user to enter trip duration
monitor_main_power_CB();
clear_LCD();

trip_duration_volt = ReadADC(0x00);   //Read trip duration ADC0
if(troubleshooting)
  {                                    //display voltage LCD
  convert_display_voltage_LCD(trip_duration_volt);
  delay(46);
  }
calculate_trip_int();
if(troubleshooting)
  {
  convert_int_to_string_display_LCD(flt_25);
  delay(46);
  }
perform_countdown();
LED_blastoff_sequence();
sound_alarm();
delay(46);
reset_alarm();
systems_A_OK();

while(1)
  {
  if(flt_timer > flt_25)  achieved25  = YES;
  if(flt_timer > flt_50)  achieved50  = YES;
  if(flt_timer > flt_75)  achieved75  = YES;
  if(flt_timer > flt_100) achieved100 = YES;

   if((achieved25==YES)&&(action25_done==NO))
        //25% flight complete
     {
```

```
     flt25_actions();
     action25_done=YES;
 systems_A_OK();
     }

   if((achieved50==YES)&&(action50_done==NO))
        //50% flight complete
     {
     flt50_actions();
     action50_done=YES;
 systems_A_OK();
     }

   if((achieved75==YES)&&(action75_done==NO))
        //75% flight complete
     {
     flt75_actions();
     action75_done=YES;
 systems_A_OK();
     }

   if((achieved100==YES)&&(action100_done==NO))
      //100% flight complete
     {
     flt100_actions();
     action100_done=YES;
     }
   }//end while
}//end main

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

//***********************************************************************
//initialize_ports: provides initial configuration for I/O ports
//
//Note: when the RSTDISBL fuse is unprogrammed, the RESET circuitry is
//      connected to the pin, and the pin can not be used as an I/O pin.
//***********************************************************************
```

```
void initialize_ports(void)
{
DDRB = 0xff;                //PORTB[7:0] as output
PORTB= 0x00;                //initialize low
DDRC = 0xb0;                //set PORTC as output  OROO_IIII 1011_0000
PORTC= PORTC_pullup_mask;   //initialize pullups PORTC[3:1]
DDRD = 0xff;                //set PORTD as output
PORTD =0x00;                //initialize low
}

//*************************************************************************
//delay(unsigned int num_of_65_5ms_interrupts): this generic delay function
//provides the specified delay as the number of 65.5 ms "clock ticks"
//from the Timer0 interrupt.
//Note: this function is only valid when using a 1 MHz crystal or ceramic
//      resonator
//*************************************************************************

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

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

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

//*************************************************************************
//ReadADC: read analog voltage from ADC-the desired channel for conversion
//is passed in as an unsigned character variable. The result is returned
//as a left justified, 10 bit binary result.
//The ADC prescalar must be set to 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 results
while (!(ADCSRA & 0x10));             //Check if conversion is ready
ADCSRA |= 0x10;                       //Clear Conv rdy flag - set the bit
binary_weighted_voltage_low = ADCL;   //Read 8 low bits first (important)
                                      //Read 2 high bits, multiply by 256
binary_weighted_voltage_high = ((unsigned int)(ADCH << 8));
binary_weighted_voltage=binary_weighted_voltage_low |
                        binary_weighted_voltage_high;
return binary_weighted_voltage;       //ADCH:ADCL
}

//*************************************************************************
//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 1 MHz and then
//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)
{
TCCR0 = 0x04; //divide timer0 timebase by 256, overfl. occurs every 65.5ms
TIMSK = 0x01; //enable timer0 overflow interrupt
asm("SEI");   //enable global interrupt
}

//*************************************************************************
//LCD_Init: initialization for an LCD connected in the following manner:
//LCD: AND671GST 1x16 character display
//LCD configured as two 8 character lines in a 1x16 array
//LCD data bus (pin 14-pin7) ATMEL 8: PORTD
//LCD RS (pin 28)  ATMEL 8: PORTC[5]
//LCD E  (pin 27)  ATMEL 8: PORTC[4]
//*************************************************************************

void LCD_Init(void)
{
delay(1);
delay(1);
delay(1);
                   // output command string to initialize LCD
putcommand(0x38);  //function set 8-bit
delay(1);
putcommand(0x38);  //function set 8-bit
putcommand(0x38);  //function set 8-bit
putcommand(0x38);  //one line, 5x7 char
putcommand(0x0C);  //display on
putcommand(0x01);  //display clear-1.64 ms
putcommand(0x06);  //entry mode set
putcommand(0x00);  //clear display, cursor at home
putcommand(0x00);  //clear display, cursor at home
}

//*************************************************************************
//putchar:prints specified ASCII character to LCD
//*************************************************************************

void putchar(unsigned char c)
```

```
{
DDRD  = 0xff;                             //set PORTD as output
DDRC  = DDRC|0x30;                       //make PORTC[5:4] output
PORTD = c;
PORTC = (PORTC|0x20)|PORTC_pullup_mask;   //RS=1
PORTC = (PORTC|0x10)|PORTC_pullup_mask;;  //E=1
PORTC = (PORTC&0xef)|PORTC_pullup_mask;;  //E=0
delay(1);
}

//*************************************************************************
//putcommand: performs specified LCD related command
//*************************************************************************

void putcommand(unsigned char d)
{
DDRD  = 0xff;                             //set PORTD as output
DDRC  = DDRC|0xC0;                        //make PORTA[5:4] output
PORTC = (PORTC&0xdf)|PORTC_pullup_mask;   //RS=0
PORTD = d;
PORTC = (PORTC|0x10)|PORTC_pullup_mask;   //E=1
PORTC = (PORTC&0xef)|PORTC_pullup_mask;   //E=0
delay(1);
}

//*************************************************************************
//clear_LCD: clears LCD
//*************************************************************************

void clear_LCD(void)
{
putcommand(0x01);
}

//*************************************************************************
//*void calculate_trip_int(void)
//*************************************************************************

void calculate_trip_int(void)
```

```
{
unsigned int    trip_duration_sec;
unsigned int    trip_duration_int;

trip_duration_sec=(unsigned int)(((double)(trip_duration_volt)/1024.0)*
                  60.0*60.0);
trip_duration_int = (unsigned int)((double)(trip_duration_sec)/0.0655);
flt_25 = (unsigned int)((double)(trip_duration_int) * 0.25);
flt_50 = (unsigned int)((double)(trip_duration_int) * 0.50);
flt_75 = (unsigned int)((double)(trip_duration_int) * 0.75);
flt_100 = trip_duration_int;
}

//*************************************************************************
//void timer0_interrupt_isr(void)
//*************************************************************************

void timer0_interrupt_isr(void)
{
delay_timer++;
flt_timer++;                    //increment flight timer

if(flash_panel==YES)
  {
  if(flash_timer <= 8)
    {
    flash_timer++;
}
  else
    {
flash_timer = 0;
if(PORTB_LEDs == OFF)
  {
  PORTB = 0xff;
  PORTB_LEDs = ON;
  }
else
  {
  PORTB = 0x00;
  PORTB_LEDs = OFF;
  }
 }
  }
else
  {
  flash_timer = 0;
  }
}

//***********************************************************************
//void flt25_actions(void)
//***********************************************************************

void flt25_actions(void)
{
sound_alarm();
flash_LED_panel();
print_LOWO2();
monitor_O2_CB_reset();
reset_alarm();
restore_panel();
action25_done = YES;
}

//***********************************************************************
//void flt50_actions(void)
//***********************************************************************

void flt50_actions(void)
{
sound_alarm();
flash_LED_panel();
print_LOW_FUEL();
monitor_aux_fuel_CB();
reset_alarm();
restore_panel();
action50_done = YES;
}
```

```
//*************************************************************************
//void flt75_actions(void)
//*************************************************************************

void flt75_actions(void)
{
sound_alarm();
flash_LED_panel();
print_OVERHEAT();
delay(458);                           //delay 30s
monitor_main_power_CB();
reset_alarm();
restore_panel();
action75_done = YES;
}

//*************************************************************************
//void flt100_actions(void)
//*************************************************************************

void flt100_actions(void)
{
sound_alarm();
flash_LED_panel();
print_fuel_expended();
turn_off_LEDs();
action100_done = YES;
}


//*************************************************************************
//void sound_alarm(void)
//*************************************************************************

void sound_alarm(void)
{
PORTB = PORTB | 0x80;
}
```

```
//*************************************************************************
//void turn_off_LEDs(void)
//*************************************************************************

void turn_off_LEDs(void)
{
PORTB = PORTB & 0x80;
}


//*************************************************************************
//void reset_alarm(void)
//*************************************************************************

void reset_alarm(void)
{
PORTB = PORTB & 0x7F;
}

//*************************************************************************
//void restore_panel(void)
//*************************************************************************

void restore_panel(void)
{
flash_panel = NO;
PORTB = PORTB | 0x7F;
}

//*************************************************************************
//void LED_blastoff_sequence(void)
//*************************************************************************

void LED_blastoff_sequence(void)
{
PORTB = 0x00;         //0000_0000
delay(15);            //delay 1s
PORTB = 0x01;         //0000_0001
```

```
delay(15);             //delay 1s
PORTB = 0x03;          //0000_0011
delay(15);             //delay 1s
PORTB = 0x07;          //0000_0111
delay(15);             //delay 1s
PORTB = 0x1F;          //0001_1111
delay(15);             //delay 1s
PORTB = 0x7F;          //0111_1111
delay(15);             //delay 1s
}

//*************************************************************************
//void LED_power_down_sequence(void)
//*************************************************************************

void LED_power_down_sequence(void)
{
PORTB = 0x7F;          //0111_1111
delay(15);             //delay 1s
PORTB = 0x1F;          //0001_1111
delay(15);             //delay 1s
PORTB = 0x07;          //0000_0111
delay(15);             //delay 1s
PORTB = 0x03;          //0000_0011
delay(15);             //delay 1s
PORTB = 0x01;          //0000_0001
delay(15);             //delay 1s
PORTB = 0x00;          //0000_0000
delay(15);             //delay 1s
}

//*************************************************************************
//void monitor_main_power_CB(void)
//*************************************************************************

void monitor_main_power_CB(void)
{
  while((PINC & 0x02) == 0x02)
  {
```

```
  ;    //wait for PC1 to be exerted low
  }
}

//*************************************************************************
//void monitor_O2_CB_reset(void)
//*************************************************************************

void monitor_O2_CB_reset(void)
{
  while((PINC & 0x08) == 0x08)
    {
    ;   //wait for PC3 to be exerted low
    }
}

//*************************************************************************
//void monitor_aux_fuel_CB(void)
//*************************************************************************

void monitor_aux_fuel_CB(void)
{
  while((PINC & 0x04) == 0x04)
    {
    ;   //wait for PC2 to be exerted low
    }
}

//*************************************************************************
//void perform_countdown(void)
//*************************************************************************

void perform_countdown(void)
{
clear_LCD();
putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('1'); putchar ('0'); //print 10
delay(15);                   //delay 1s
```

```
putcommand(0x01);               //cursor home
putcommand(0x80);               //DD RAM location 1 - line 1
putchar('9');                   //print 9
delay(15);                      //delay 1s

putcommand(0x01);               //cursor home
putcommand(0x80);               //DD RAM location 1 - line 1
putchar('8');                   //print 8
delay(15);                      //delay 1s

putcommand(0x01);               //cursor home
putcommand(0x80);               //DD RAM location 1 - line 1
putchar('7');                   //print 7
delay(15);                      //delay 1s

putcommand(0x01);               //cursor home
putcommand(0x80);               //DD RAM location 1 - line 1
putchar('6');                   //print 6
delay(15);                      //delay 1s

putcommand(0x01);               //cursor home
putcommand(0x80);               //DD RAM location 1 - line 1
putchar('5');                   //print 5
delay(15);                      //delay 1s

putcommand(0x01);               //cursor home
putcommand(0x80);               //DD RAM location 1 - line 1
putchar('4');                   //print 4
delay(15);                      //delay 1s

putcommand(0x01);               //cursor home
putcommand(0x80);               //DD RAM location 1 - line 1
putchar('3');                   //print 3
delay(15);                      //delay 1s

putcommand(0x01);               //cursor home

putcommand(0x80);               //DD RAM location 1 - line 1
```

```
putchar('2');                 //print 2
delay(15);                    //delay 1s

putcommand(0x01);             //cursor home
putcommand(0x80);             //DD RAM location 1 - line 1
putchar('1');                 //print 1
delay(15);                    //delay 1s

putcommand(0x01);             //cursor home
putcommand(0x80);             //DD RAM location 1 - line 1
putchar('0');                 //print 0
delay(15);                    //delay 1s


//BLASTOFF!
putcommand(0x01);             //cursor home
putcommand(0x80);             //DD RAM location 1 - line 1
putchar('B'); putchar('L'); putchar('A'); putchar('S'); putchar('T');
putchar('O'); putchar('F'); putchar('F'); putchar('!');

}

//*************************************************************************
//void print_LOWO2(void)
//*************************************************************************

void print_LOWO2(void)
{
clear_LCD();
putcommand(0x01);             //cursor home
putcommand(0x80);             //DD RAM location 1 - line 1
putchar('L'); putchar('O'); putchar('W'); putchar(' '); putchar('O');
putchar('2');

putcommand(0xC0);//DD RAM location 1 - line 2
putchar('R'); putchar('E'); putchar('S'); putchar('E'); putchar('T');
putchar(' '); putchar('O'); putchar('2'); putchar(' '); putchar('C');
putchar('B');
}
```

```
//*************************************************************************
//void print_LOW_FUEL(void)
//*************************************************************************

void print_LOW_FUEL(void)
{
clear_LCD();
putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('L'); putchar('O'); putchar('W'); putchar(' '); putchar('F');
putchar('U'); putchar('E'); putchar('L');

putcommand(0xC0);//DD RAM location 1 - line 2
putchar('A'); putchar('S'); putchar('S'); putchar('E'); putchar('R');
putchar('T'); putchar(' '); putchar('A'); putchar('U'); putchar('X');
putchar('F'); putchar('U'); putchar('E'); putchar('L'); putchar('C');
putchar('B');
}

//*************************************************************************
//void print_fuel_expended(void)
//*************************************************************************

void print_fuel_expended(void)
{
clear_LCD();
putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('F'); putchar('U'); putchar('E'); putchar('L'); putchar(' ');
putchar('E'); putchar('X'); putchar('P'); putchar('E'); putchar('N');
putchar('D'); putchar('E'); putchar('D');

putcommand(0xC0);//DD RAM location 1 - line 2
putchar('S'); putchar('H'); putchar('U'); putchar('T'); putchar('T');
putchar('I'); putchar('N'); putchar('G'); putchar(' '); putchar('D');
putchar('O'); putchar('W'); putchar('N'); putchar('.'); putchar('.');
putchar('.');
}
```

```
//***************************************************************************
//void print_trip_dur(void);
//***************************************************************************

void print_trip_dur(void)
{
clear_LCD();
putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('T'); putchar('R'); putchar('I'); putchar('P');
putchar('T'); putchar('I'); putchar('M'); putchar('E'); putchar(':');
putchar('0'); putchar('-'); putchar('6'); putchar('0');

putcommand(0xC0);//DD RAM location 1 - line 2
putchar('S'); putchar('E'); putchar('T'); putchar(' '); putchar('M');
putchar('A'); putchar('I'); putchar('N'); putchar(' '); putchar('P');
putchar('W'); putchar('R'); putchar(' '); putchar('C'); putchar('B');
}


//***************************************************************************
//void print_OVERHEAT(void)
//***************************************************************************

void print_OVERHEAT(void)
{
clear_LCD();
putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('E'); putchar('N'); putchar('G'); putchar('I'); putchar('N');
putchar('E'); putchar(' '); putchar('O'); putchar('V'); putchar('E');
putchar('R'); putchar('H'); putchar('E'); putchar('A'); putchar('T');

putcommand(0xC0);//DD RAM location 1 - line 2
putchar('R'); putchar('E'); putchar('S'); putchar('E'); putchar('T');
putchar(' '); putchar('E'); putchar('N'); putchar('G'); putchar(' ');
putchar('C'); putchar('B'); putchar(' '); putchar('3'); putchar('0');
putchar('S');
```

```
}

//*************************************************************************
//void systems_A_OK(void)
//*************************************************************************

void systems_A_OK(void)
{
clear_LCD();
putcommand(0x01);            //cursor home
putcommand(0x80);            //DD RAM location 1 - line 1
putchar('S'); putchar('Y'); putchar('S'); putchar('T'); putchar('E');
putchar('M'); putchar('S'); putchar(' '); putchar('A'); putchar('-');
putchar('O'); putchar('K'); putchar('!'); putchar('!'); putchar('!');
}

//*************************************************************************
//void flash_LED_panel(void)
//*************************************************************************

void flash_LED_panel(void)
{
flash_panel = YES;
flash_timer = 0;
PORTB = 0x00;
PORTB_LEDs = OFF;
}

//*************************************************************************
//convert_display_voltage_LCD: converts binary weighted voltage to ASCII
//representation and prints result to LCD screen
//*************************************************************************

void convert_display_voltage_LCD(int binary_voltage)
{
float actual_voltage;                  //voltage between 0 and 5 volts
int   all_integer_voltage;
 //integer representation of voltage
                                       //int representation of voltage
```

```
int   hundreths_place, tens_place, ones_place;
                                       //char representation of voltage
char  hundreths_place_char, tens_place_char, ones_place_char;

                                        // display analog voltage on LCD
putcommand(0xC0);                       //LCD cursor to line 2
                                        //scale float voltage 0..5V
actual_voltage = ((float)(binary_voltage)/(float)(0x3FF))*5.0;
                                        //voltage represented 0 to 500
all_integer_voltage=actual_voltage * 100;//represent as all integer
hundreths_place = all_integer_voltage/100;//isolate first digit
hundreths_place_char = (char)(hundreths_place + 48); //convert to ascii
putchar(hundreths_place_char);          //display first digit
putchar('.');                           //print decimal point to LCD
                                        //isolate tens place
tens_place = (int)((all_integer_voltage - (hundreths_place*100))/10);
tens_place_char=(char)(tens_place+48);  //convert to ASCII
putchar(tens_place_char);               //print to LCD
                                        //isolate ones place
ones_place = (int)((all_integer_voltage - (hundreths_place*100))%10);
ones_place_char=(char)(ones_place+48);  //convert to ASCII
putchar(ones_place_char);               //print to LCD
putchar('V');                           //print unit V
}

//***************************************************************************
//convert_int_to_string_display_LCD: converts 16 bit to unsigned integer
//values range from 0 to 65,535
//prints result to LCD screen
//***************************************************************************

void convert_int_to_string_display_LCD(unsigned int total_integer_value)
{
int   ten_thousandths_place, thousandths_place;
int   hundreths_place, tens_place, ones_place;
char  ten_thousandths_place_char, thousandths_place_char;
char  hundreths_place_char, tens_place_char, ones_place_char;

putcommand(0xC0);                       //LCD cursor to line 2
```

```
                                        //10,000th place
ten_thousandths_place = total_integer_value/10000;
ten_thousandths_place_char = (char)(ten_thousandths_place+48);
putchar(ten_thousandths_place_char);
                                        //1,000th place
thousandths_place = (int)((total_integer_value -
                          (ten_thousandths_place*10000))/1000);
thousandths_place_char = (char)(thousandths_place+48);
putchar(thousandths_place_char);
                                       //100th place
hundreths_place = (int)((total_integer_value -
                        (ten_thousandths_place*10000)-
(thousandths_place*1000))/100);
hundreths_place_char = (char)(hundreths_place + 48);
putchar(hundreths_place_char);
                                        //10th place
tens_place = (int)((total_integer_value -(ten_thousandths_place*10000)-
   (thousandths_place*1000)-(hundreths_place*100))/10);
tens_place_char=(char)(tens_place+48);  //convert to ASCII
putchar(tens_place_char);               //print to LCD
                                        //isolate ones place
ones_place = (int)((total_integer_value -(ten_thousandths_place*10000)-
   (thousandths_place*1000)-(hundreths_place*100))%10);
ones_place_char=(char)(ones_place+48);  //convert to ASCII
putchar(ones_place_char);               //print to LCD
}


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

8.12 SUMMARY

In this chapter, we discussed the voltage and current operating parameters for the Arduino Duemilanove processing board and the Atmel ATmega328 microcontroller. We discussed how this information may be applied to properly design an interface for common input and output circuits. It must be emphasized a properly designed interface allows the microcontroller to operate properly within its parameter envelope. If due to a poor interface design, a microcontroller is used outside its prescribed operating parameter values, spurious and incorrect logic values will result. We provided interface information for a wide range of input and output devices. We also discussed the concept of interfacing a motor to a microcontroller using PWM techniques coupled with high power MOSFET or SSR switching devices. We closed the chapter with three extended examples.

8.13 REFERENCES

- Pack D, Barrett S (2002) 68HC12 Microcontroller: Theory and Applications. Prentice-Hall Incorporated, Upper Saddle River, NJ.
- Barrett S, Pack D (2004) Embedded Systems Design with the 68HC12 and HCS12. Prentice-Hall Incorporated, Upper Saddle River, NJ.
- Crydom Corporation, 2320 Paseo de las Americas, Suite 201, San Diego, CA (`www.crydom.com`).
- Sick/Stegmann Incorporated, Dayton, OH, (`www.stegmann.com`).
- Images Company, 39 Seneca Loop, Staten Island, NY 10314.
- *Atmel 8-bit AVR Microcontroller with 16/32/64K Bytes In-System Programmable Flash, ATmega328P/V, ATmega324P/V, 644P/V* data sheet: 8011I-AVR-05/08, Atmel Corporation, 2325 Orchard Parkway, San Jose, CA 95131.
- Barrett S, Pack D (2006) Microcontrollers Fundamentals for Engineers and Scientists. Morgan and Claypool Publishers. DOI: 10.2200/S00025ED1V01Y200605DCS001
- Barrett S and Pack D (2008) Atmel AVR Microcontroller Primer Programming and Interfacing. Morgan and Claypool Publishers. DOI: 10.2200/S00100ED1V01Y200712DCS015
- Barrett S (2010) Embedded Systems Design with the Atmel AVR Microcontroller. Morgan and Claypool Publishers. DOI: 10.2200/S00225ED1V01Y200910DCS025
- National Semiconductor, *LM34/LM34A/LM34C/LM34CA/LM34D Precision Fahrenheit Temperature Sensor*, 1995.

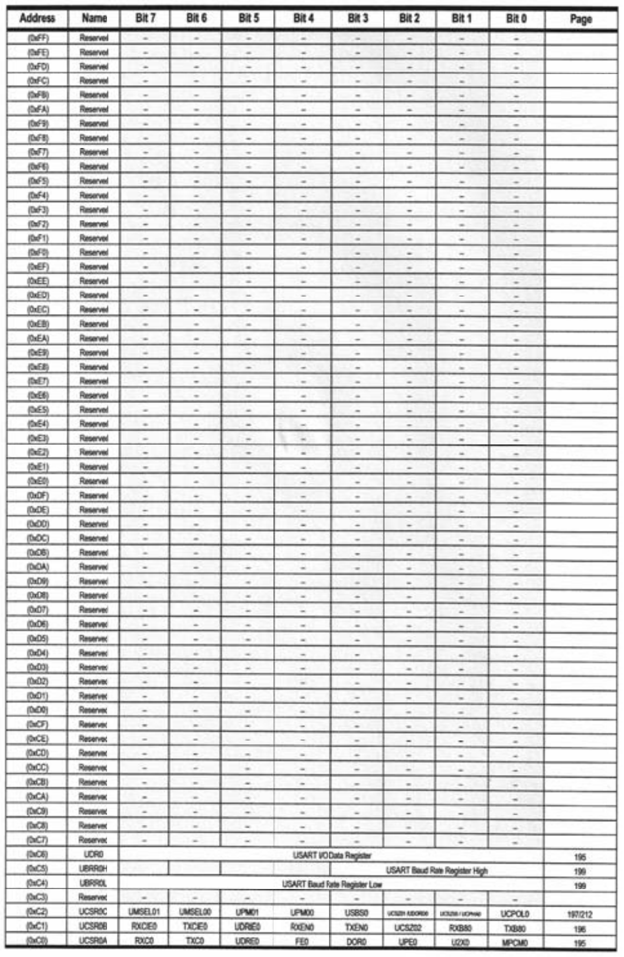
8.14 CHAPTER PROBLEMS

1. What will happen if a microcontroller is used outside of its prescribed operating envelope?
2. Discuss the difference between the terms "sink" and "source" as related to current loading of a microcontroller.
3. Can an LED with a series limiting resistor be directly driven by the Atmel microcontroller? Explain.

4. In your own words, provide a brief description of each of the microcontroller electrical parameters.
5. What is switch bounce? Describe two techniques to minimize switch bounce.
6. Describe a method of debouncing a keypad.
7. What is the difference between an incremental encoder and an absolute encoder? Describe applications for each type.
8. What must be the current rating of the 2N2222 and 2N2907 transistors used in the tri-state LED circuit? Support your answer.
9. Draw the circuit for a six character seven segment display. Fully specify all components. Write a program to display "ATmega328. "
10. Repeat the question above for a dot matrix display.
11. Repeat the question above for a LCD display.
12. What is the difference between a unipolar and bipolar stepper motor?
13. What controls the speed of rotation of a stepper motor?
14. A stepper motor provides and angular displacement of 1.8 degrees per step. How can this resolution be improved?
15. Write a function to convert an ASCII numeral representation (0 to 9) to a seven segment display.
16. Why is an interface required between a microcontroller and a stepper motor?

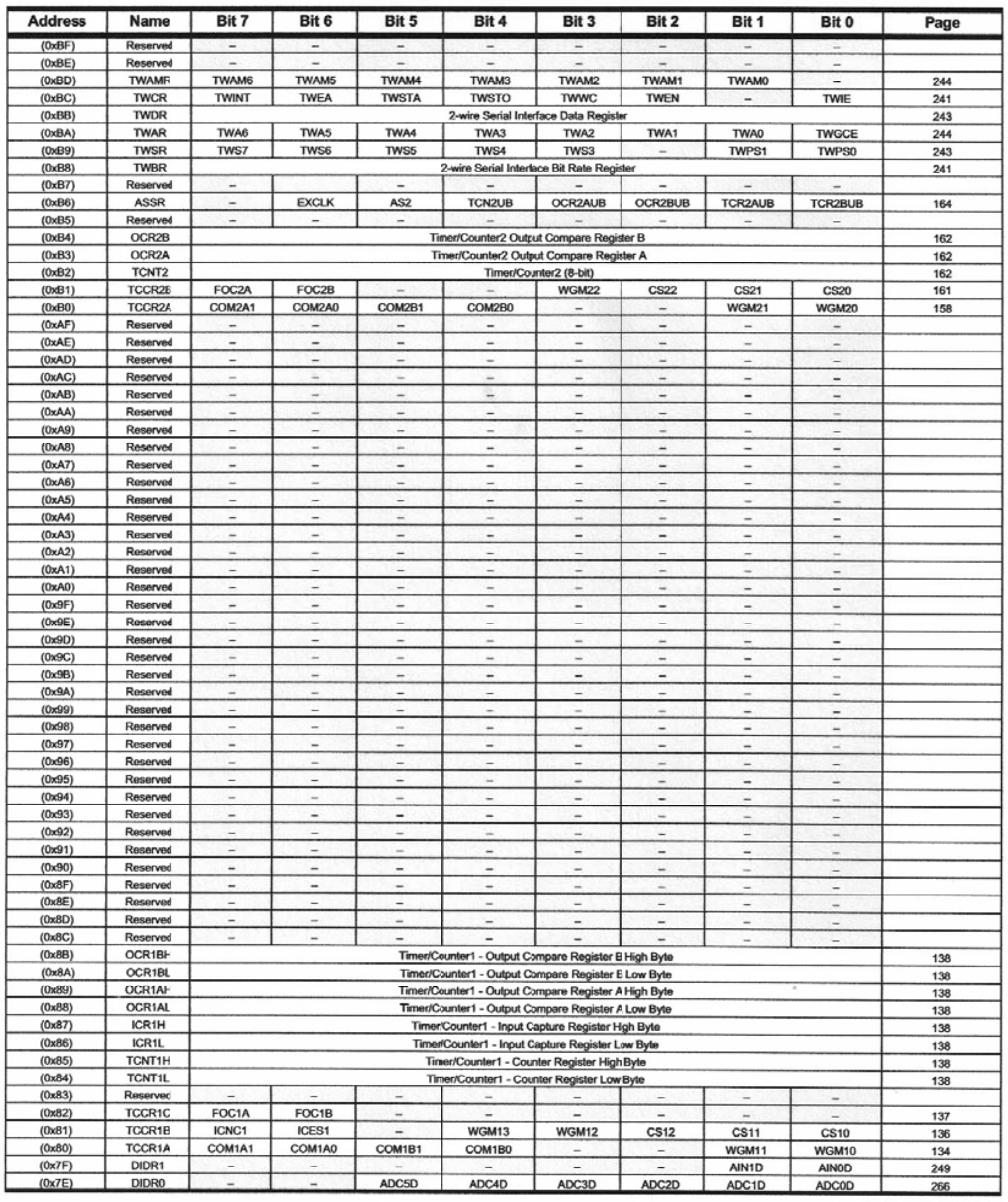
APPENDIX A

ATmega328 Register Set



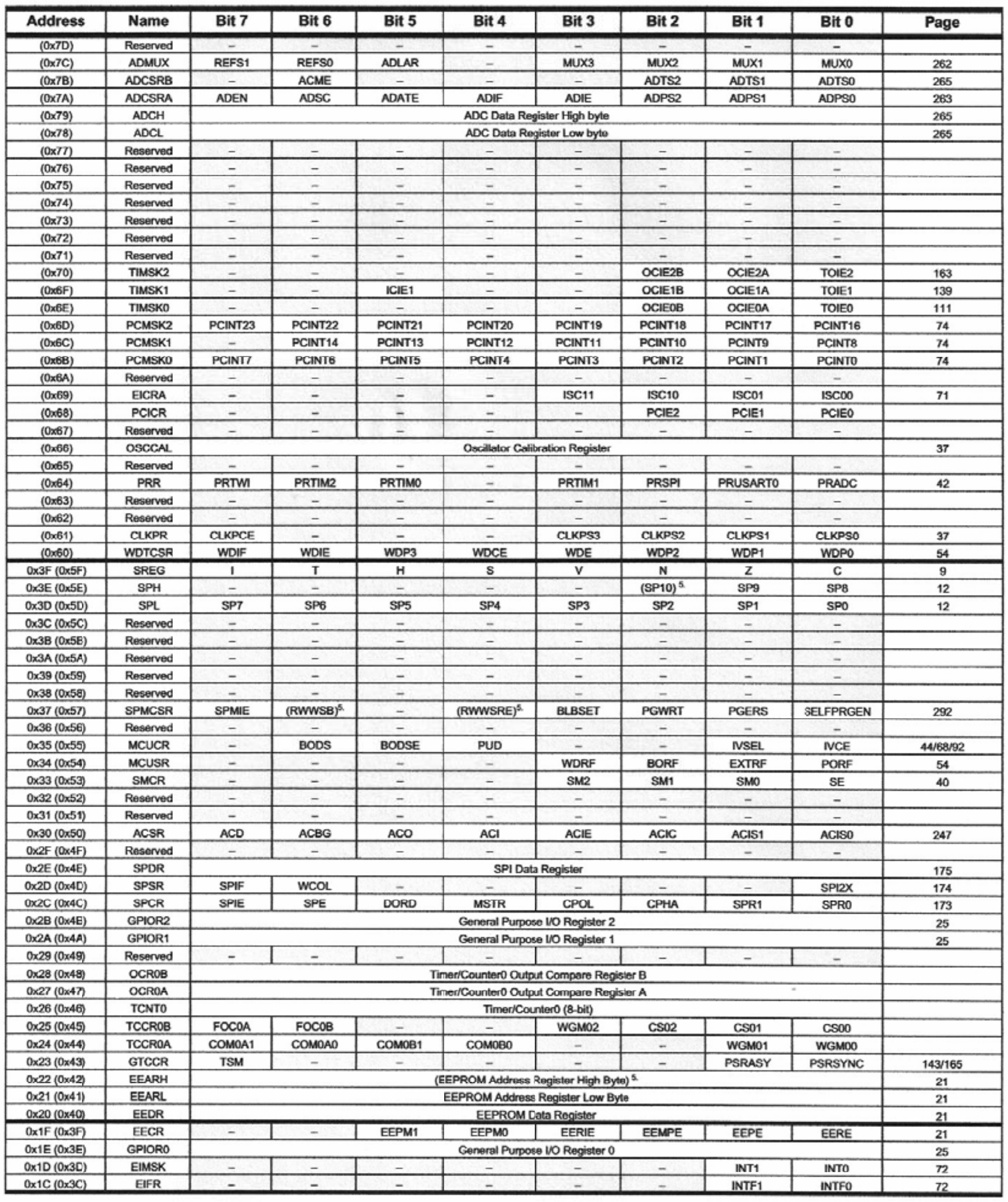
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(0xFF)	Reserved	–	–	–	–	–	–	–	–	
(0xFE)	Reserved	–	–	–	–	–	–	–	–	
(0xFD)	Reserved	–	–	–	–	–	–	–	–	
(0xFC)	Reserved	–	–	–	–	–	–	–	–	
(0xFB)	Reserved	–	–	–	–	–	–	–	–	
(0xFA)	Reserved	–	–	–	–	–	–	–	–	
(0xF9)	Reserved	–	–	–	–	–	–	–	–	
(0xF8)	Reserved	–	–	–	–	–	–	–	–	
(0xF7)	Reserved	–	–	–	–	–	–	–	–	
(0xF6)	Reserved	–	–	–	–	–	–	–	–	
(0xF5)	Reserved	–	–	–	–	–	–	–	–	
(0xF4)	Reserved	–	–	–	–	–	–	–	–	
(0xF3)	Reserved	–	–	–	–	–	–	–	–	
(0xF2)	Reserved	–	–	–	–	–	–	–	–	
(0xF1)	Reserved	–	–	–	–	–	–	–	–	
(0xF0)	Reserved	–	–	–	–	–	–	–	–	
(0xEF)	Reserved	–	–	–	–	–	–	–	–	
(0xEE)	Reserved	–	–	–	–	–	–	–	–	
(0xED)	Reserved	–	–	–	–	–	–	–	–	
(0xEC)	Reserved	–	–	–	–	–	–	–	–	
(0xEB)	Reserved	–	–	–	–	–	–	–	–	
(0xEA)	Reserved	–	–	–	–	–	–	–	–	
(0xE9)	Reserved	–	–	–	–	–	–	–	–	
(0xE8)	Reserved	–	–	–	–	–	–	–	–	
(0xE7)	Reserved	–	–	–	–	–	–	–	–	
(0xE6)	Reserved	–	–	–	–	–	–	–	–	
(0xE5)	Reserved	–	–	–	–	–	–	–	–	
(0xE4)	Reserved	–	–	–	–	–	–	–	–	
(0xE3)	Reserved	–	–	–	–	–	–	–	–	
(0xE2)	Reserved	–	–	–	–	–	–	–	–	
(0xE1)	Reserved	–	–	–	–	–	–	–	–	
(0xE0)	Reserved	–	–	–	–	–	–	–	–	
(0xDF)	Reserved	–	–	–	–	–	–	–	–	
(0xDE)	Reserved	–	–	–	–	–	–	–	–	
(0xDD)	Reserved	–	–	–	–	–	–	–	–	
(0xDC)	Reserved	–	–	–	–	–	–	–	–	
(0xDB)	Reserved	–	–	–	–	–	–	–	–	
(0xDA)	Reserved	–	–	–	–	–	–	–	–	
(0xD9)	Reserved	–	–	–	–	–	–	–	–	
(0xD8)	Reserved	–	–	–	–	–	–	–	–	
(0xD7)	Reserved	–	–	–	–	–	–	–	–	
(0xD6)	Reserved	–	–	–	–	–	–	–	–	
(0xD5)	Reserved	–	–	–	–	–	–	–	–	
(0xD4)	Reserved	–	–	–	–	–	–	–	–	
(0xD3)	Reserved	–	–	–	–	–	–	–	–	
(0xD2)	Reserved	–	–	–	–	–	–	–	–	
(0xD1)	Reserved	–	–	–	–	–	–	–	–	
(0xD0)	Reserved	–	–	–	–	–	–	–	–	
(0xCF)	Reserved	–	–	–	–	–	–	–	–	
(0xCE)	Reserved	–	–	–	–	–	–	–	–	
(0xCD)	Reserved	–	–	–	–	–	–	–	–	
(0xCC)	Reserved	–	–	–	–	–	–	–	–	
(0xCB)	Reserved	–	–	–	–	–	–	–	–	
(0xCA)	Reserved	–	–	–	–	–	–	–	–	
(0xC9)	Reserved	–	–	–	–	–	–	–	–	
(0xC8)	Reserved	–	–	–	–	–	–	–	–	
(0xC7)	Reserved	–	–	–	–	–	–	–	–	
(0xC6)	UDR0	USART I/O Data Register								195
(0xC5)	UBRR0H					USART Baud Rate Register High				199
(0xC4)	UBRR0L	USART Baud Rate Register Low								199
(0xC3)	Reserved	–	–	–	–	–	–	–	–	
(0xC2)	UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01 /UDORD0	UCSZ00 / UCPHA0	UCPOL0	197/212
(0xC1)	UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ02	RXB80	TXB80	196
(0xC0)	UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0	195

Figure A.1: Atmel AVR ATmega328 Register Set. (Figure used with permission of Atmel, Incorporated.)



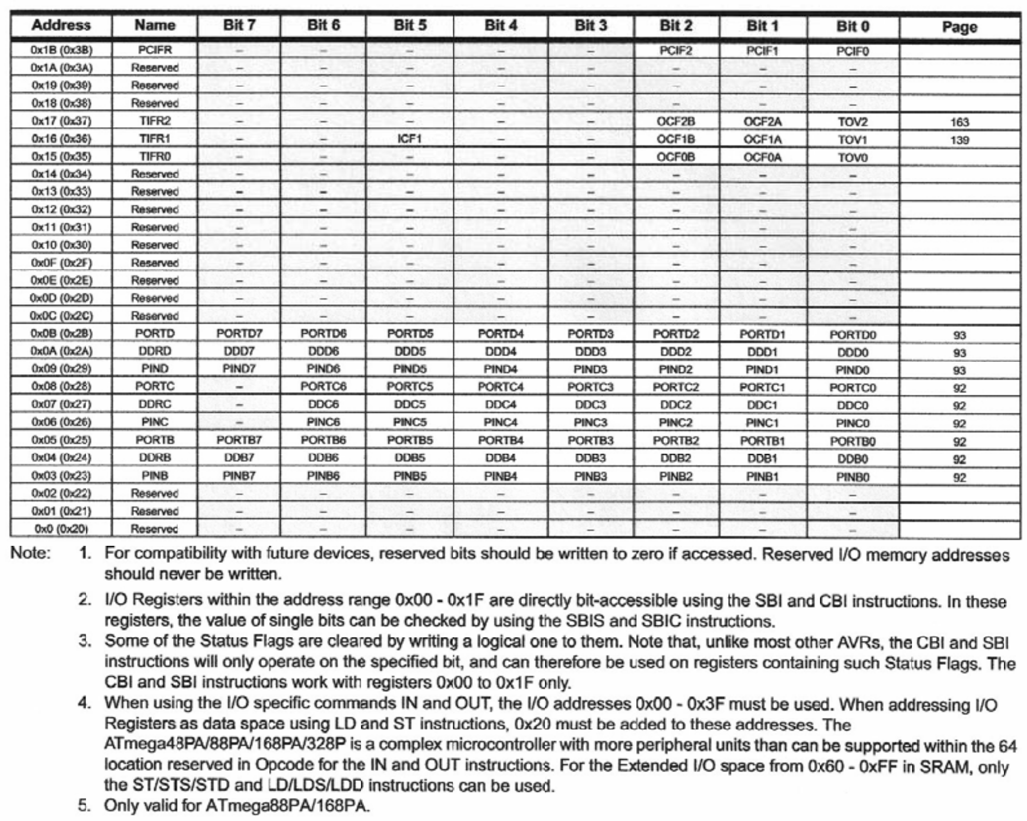
Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(0xBF)	Reserved	–	–	–	–	–	–	–	–	
(0xBE)	Reserved	–	–	–	–	–	–	–	–	
(0xBD)	TWAMR	TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	–	244
(0xBC)	TWCR	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	241
(0xBB)	TWDR	2-wire Serial Interface Data Register								243
(0xBA)	TWAR	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE	244
(0xB9)	TWSR	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0	243
(0xB8)	TWBR	2-wire Serial Interface Bit Rate Register								241
(0xB7)	Reserved	–		–	–	–	–	–	–	
(0xB6)	ASSR	–	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB	164
(0xB5)	Reserved	–	–	–	–	–	–	–	–	
(0xB4)	OCR2B	Timer/Counter2 Output Compare Register B								162
(0xB3)	OCR2A	Timer/Counter2 Output Compare Register A								162
(0xB2)	TCNT2	Timer/Counter2 (8-bit)								162
(0xB1)	TCCR2B	FOC2A	FOC2B	–	–	WGM22	CS22	CS21	CS20	161
(0xB0)	TCCR2A	COM2A1	COM2A0	COM2B1	COM2B0	–	–	WGM21	WGM20	158
(0xAF)	Reserved	–	–	–	–	–	–	–	–	
(0xAE)	Reserved	–	–	–	–	–	–	–	–	
(0xAD)	Reserved	–	–	–	–	–	–	–	–	
(0xAC)	Reserved	–	–	–	–	–	–	–	–	
(0xAB)	Reserved	–	–	–	–	–	–	–	–	
(0xAA)	Reserved	–	–	–	–	–	–	–	–	
(0xA9)	Reserved	–	–	–	–	–	–	–	–	
(0xA8)	Reserved	–	–	–	–	–	–	–	–	
(0xA7)	Reserved	–	–	–	–	–	–	–	–	
(0xA6)	Reserved	–	–	–	–	–	–	–	–	
(0xA5)	Reserved	–	–	–	–	–	–	–	–	
(0xA4)	Reserved	–	–	–	–	–	–	–	–	
(0xA3)	Reserved	–	–	–	–	–	–	–	–	
(0xA2)	Reserved	–	–	–	–	–	–	–	–	
(0xA1)	Reserved	–	–	–	–	–	–	–	–	
(0xA0)	Reserved	–	–	–	–	–	–	–	–	
(0x9F)	Reserved	–	–	–	–	–	–	–	–	
(0x9E)	Reserved	–	–	–	–	–	–	–	–	
(0x9D)	Reserved	–	–	–	–	–	–	–	–	
(0x9C)	Reserved	–	–	–	–	–	–	–	–	
(0x9B)	Reserved	–	–	–	–	–	–	–	–	
(0x9A)	Reserved	–	–	–	–	–	–	–	–	
(0x99)	Reserved	–	–	–	–	–	–	–	–	
(0x98)	Reserved	–	–	–	–	–	–	–	–	
(0x97)	Reserved	–	–	–	–	–	–	–	–	
(0x96)	Reserved	–	–	–	–	–	–	–	–	
(0x95)	Reserved	–	–	–	–	–	–	–	–	
(0x94)	Reserved	–	–	–	–	–	–	–	–	
(0x93)	Reserved	–	–	–	–	–	–	–	–	
(0x92)	Reserved	–	–	–	–	–	–	–	–	
(0x91)	Reserved	–	–	–	–	–	–	–	–	
(0x90)	Reserved	–	–	–	–	–	–	–	–	
(0x8F)	Reserved	–	–	–	–	–	–	–	–	
(0x8E)	Reserved	–	–	–	–	–	–	–	–	
(0x8D)	Reserved	–	–	–	–	–	–	–	–	
(0x8C)	Reserved	–	–	–	–	–	–	–	–	
(0x8B)	OCR1BH	Timer/Counter1 - Output Compare Register B High Byte								138
(0x8A)	OCR1BL	Timer/Counter1 - Output Compare Register B Low Byte								138
(0x89)	OCR1AH	Timer/Counter1 - Output Compare Register A High Byte								138
(0x88)	OCR1AL	Timer/Counter1 - Output Compare Register A Low Byte								138
(0x87)	ICR1H	Timer/Counter1 - Input Capture Register Hgh Byte								138
(0x86)	ICR1L	Timer/Counter1 - Input Capture Register Low Byte								138
(0x85)	TCNT1H	Timer/Counter1 - Counter Register High Byte								138
(0x84)	TCNT1L	Timer/Counter1 - Counter Register Low Byte								138
(0x83)	Reserved	–	–	–	–	–	–	–	–	
(0x82)	TCCR1C	FOC1A	FOC1B	–	–	–	–	–	–	137
(0x81)	TCCR1B	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	136
(0x80)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	134
(0x7F)	DIDR1	–	–	–	–	–	–	AIN1D	AIN0D	249
(0x7E)	DIDR0	–	–	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D	266

Figure A.2: Atmel AVR ATmega328 Register Set. (Figure used with permission of Atmel, Incorporated.)



Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(0x7D)	Reserved	–	–	–	–	–	–	–	–	
(0x7C)	ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	262
(0x7B)	ADCSRB	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0	265
(0x7A)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	263
(0x79)	ADCH	ADC Data Register High byte								265
(0x78)	ADCL	ADC Data Register Low byte								265
(0x77)	Reserved	–	–	–	–	–	–	–	–	
(0x76)	Reserved	–	–	–	–	–	–	–	–	
(0x75)	Reserved	–	–	–	–	–	–	–	–	
(0x74)	Reserved	–	–	–	–	–	–	–	–	
(0x73)	Reserved	–	–	–	–	–	–	–	–	
(0x72)	Reserved	–	–	–	–	–	–	–	–	
(0x71)	Reserved	–	–	–	–	–	–	–	–	
(0x70)	TIMSK2	–	–	–	–	–	OCIE2B	OCIE2A	TOIE2	163
(0x6F)	TIMSK1	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	139
(0x6E)	TIMSK0	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	111
(0x6D)	PCMSK2	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	74
(0x6C)	PCMSK1	–	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	74
(0x6B)	PCMSK0	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	74
(0x6A)	Reserved	–	–	–	–	–	–	–	–	
(0x69)	EICRA	–	–	–	–	ISC11	ISC10	ISC01	ISC00	71
(0x68)	PCICR	–	–	–	–	–	PCIE2	PCIE1	PCIE0	
(0x67)	Reserved	–	–	–	–	–	–	–	–	
(0x66)	OSCCAL	Oscillator Calibration Register								37
(0x65)	Reserved	–	–	–	–	–	–	–	–	
(0x64)	PRR	PRTWI	PRTIM2	PRTIM0	–	PRTIM1	PRSPI	PRUSART0	PRADC	42
(0x63)	Reserved	–	–	–	–	–	–	–	–	
(0x62)	Reserved	–	–	–	–	–	–	–	–	
(0x61)	CLKPR	CLKPCE	–	–	–	CLKPS3	CLKPS2	CLKPS1	CLKPS0	37
(0x60)	WDTCSR	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	54
0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C	9
0x3E (0x5E)	SPH	–	–	–	–	–	(SP10)[5.]	SP9	SP8	12
0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	12
0x3C (0x5C)	Reserved	–	–	–	–	–	–	–	–	
0x3B (0x5B)	Reserved	–	–	–	–	–	–	–	–	
0x3A (0x5A)	Reserved	–	–	–	–	–	–	–	–	
0x39 (0x59)	Reserved	–	–	–	–	–	–	–	–	
0x38 (0x58)	Reserved	–	–	–	–	–	–	–	–	
0x37 (0x57)	SPMCSR	SPMIE	(RWWSB)[5.]	–	(RWWSRE)[5.]	BLBSET	PGWRT	PGERS	SELFPRGEN	292
0x36 (0x56)	Reserved	–	–	–	–	–	–	–	–	
0x35 (0x55)	MCUCR	–	BODS	BODSE	PUD	–	–	IVSEL	IVCE	44/68/92
0x34 (0x54)	MCUSR	–	–	–	–	WDRF	BORF	EXTRF	PORF	54
0x33 (0x53)	SMCR	–	–	–	–	SM2	SM1	SM0	SE	40
0x32 (0x52)	Reserved	–	–	–	–	–	–	–	–	
0x31 (0x51)	Reserved	–	–	–	–	–	–	–	–	
0x30 (0x50)	ACSR	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	247
0x2F (0x4F)	Reserved	–	–	–	–	–	–	–	–	
0x2E (0x4E)	SPDR	SPI Data Register								175
0x2D (0x4D)	SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X	174
0x2C (0x4C)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	173
0x2B (0x4B)	GPIOR2	General Purpose I/O Register 2								25
0x2A (0x4A)	GPIOR1	General Purpose I/O Register 1								25
0x29 (0x49)	Reserved	–	–	–	–	–	–	–	–	
0x28 (0x48)	OCR0B	Timer/Counter0 Output Compare Register B								
0x27 (0x47)	OCR0A	Timer/Counter0 Output Compare Register A								
0x26 (0x46)	TCNT0	Timer/Counter0 (8-bit)								
0x25 (0x45)	TCCR0B	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	
0x24 (0x44)	TCCR0A	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	
0x23 (0x43)	GTCCR	TSM	–	–	–	–	–	PSRASY	PSRSYNC	143/165
0x22 (0x42)	EEARH	(EEPROM Address Register High Byte)[5.]								21
0x21 (0x41)	EEARL	EEPROM Address Register Low Byte								21
0x20 (0x40)	EEDR	EEPROM Data Register								21
0x1F (0x3F)	EECR	–	–	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	21
0x1E (0x3E)	GPIOR0	General Purpose I/O Register 0								25
0x1D (0x3D)	EIMSK	–	–	–	–	–	–	INT1	INT0	72
0x1C (0x3C)	EIFR	–	–	–	–	–	–	INTF1	INTF0	72

Figure A.3: Atmel AVR ATmega328 Register Set. (Figure used with permission of Atmel, Incorporated.)



Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
0x1B (0x3B)	PCIFR	–	–	–	–	–	PCIF2	PCIF1	PCIF0	
0x1A (0x3A)	Reserved	–	–	–	–	–	–	–	–	
0x19 (0x39)	Reserved	–	–	–	–	–	–	–	–	
0x18 (0x38)	Reserved	–	–	–	–	–	–	–	–	
0x17 (0x37)	TIFR2	–	–	–	–	–	OCF2B	OCF2A	TOV2	163
0x16 (0x36)	TIFR1	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	139
0x15 (0x35)	TIFR0	–	–	–	–	–	OCF0B	OCF0A	TOV0	
0x14 (0x34)	Reserved	–	–	–	–	–	–	–	–	
0x13 (0x33)	Reserved	–	–	–	–	–	–	–	–	
0x12 (0x32)	Reserved	–	–	–	–	–	–	–	–	
0x11 (0x31)	Reserved	–	–	–	–	–	–	–	–	
0x10 (0x30)	Reserved	–	–	–	–	–	–	–	–	
0x0F (0x2F)	Reserved	–	–	–	–	–	–	–	–	
0x0E (0x2E)	Reserved	–	–	–	–	–	–	–	–	
0x0D (0x2D)	Reserved	–	–	–	–	–	–	–	–	
0x0C (0x2C)	Reserved	–	–	–	–	–	–	–	–	
0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	93
0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	93
0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	93
0x08 (0x28)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	92
0x07 (0x27)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	92
0x06 (0x26)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	92
0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	92
0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	92
0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	92
0x02 (0x22)	Reserved	–	–	–	–	–	–	–	–	
0x01 (0x21)	Reserved	–	–	–	–	–	–	–	–	
0x0 (0x20)	Reserved	–	–	–	–	–	–	–	–	

Note:
1. For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.
2. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions.
3. Some of the Status Flags are cleared by writing a logical one to them. Note that, unlike most other AVRs, the CBI and SBI instructions will only operate on the specified bit, and can therefore be used on registers containing such Status Flags. The CBI and SBI instructions work with registers 0x00 to 0x1F only.
4. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses. The ATmega48PA/88PA/168PA/328P is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.
5. Only valid for ATmega88PA/168PA.

Figure A.4: Atmel AVR ATmega328 Register Set. (Figure used with permission of Atmel, Incorporated.)

APPENDIX B

ATmega328 Header File

During C programming, the contents of a specific register may be referred to by name when an appropriate header file is included within your program. The header file provides the link between the register name used within a program and the hardware location of the register.

Provided below is the ATmega328 header file from the ICC AVR compiler. This header file was provided courtesy of ImageCraft Incorporated.

```
#ifndef __iom328pv_h
#define __iom328pv_h

/* ATmega328P header file for
 * ImageCraft ICCAVR compiler
 */

/* i/o register addresses
 * >= 0x60 are memory mapped only
 */

/* 2006/10/01 created
 */

/* Port D */
#define PIND (*(volatile unsigned char *)0x29)
#define DDRD (*(volatile unsigned char *)0x2A)
#define PORTD (*(volatile unsigned char *)0x2B)

/* Port C */
#define PINC (*(volatile unsigned char *)0x26)
#define DDRC (*(volatile unsigned char *)0x27)
#define PORTC (*(volatile unsigned char *)0x28)

/* Port B */
#define PINB (*(volatile unsigned char *)0x23)
#define DDRB (*(volatile unsigned char *)0x24)
```

```
#define PORTB (*(volatile unsigned char *)0x25)

/* Port A */
#define PINA (*(volatile unsigned char *)0x20)
#define DDRA (*(volatile unsigned char *)0x21)
#define PORTA (*(volatile unsigned char *)0x22)

/* Timer/Counter Interrupts */
#define TIFR0 (*(volatile unsigned char *)0x35)
#define  OCF0B    2
#define  OCF0A    1
#define  TOV0     0
#define TIMSK0 (*(volatile unsigned char *)0x6E)
#define  OCIE0B   2
#define  OCIE0A   1
#define  TOIE0    0
#define TIFR1 (*(volatile unsigned char *)0x36)
#define  ICF1     5
#define  OCF1B    2
#define  OCF1A    1
#define  TOV1     0
#define TIMSK1 (*(volatile unsigned char *)0x6F)
#define  ICIE1    5
#define  OCIE1B   2
#define  OCIE1A   1
#define  TOIE1    0
#define TIFR2 (*(volatile unsigned char *)0x37)
#define  OCF2B    2
#define  OCF2A    1
#define  TOV2     0
#define TIMSK2 (*(volatile unsigned char *)0x70)
#define  OCIE2B   2
#define  OCIE2A   1
#define  TOIE2    0

/* External Interrupts */
#define EIFR (*(volatile unsigned char *)0x3C)
#define  INTF2    2
#define  INTF1    1
```

```
#define  INTF0    0
#define EIMSK (*(volatile unsigned char *)0x3D)
#define  INT2     2
#define  INT1     1
#define  INT0     0
#define EICRA (*(volatile unsigned char *)0x69)
#define  ISC21    5
#define  ISC20    4
#define  ISC11    3
#define  ISC10    2
#define  ISC01    1
#define  ISC00    0

/* Pin Change Interrupts */
#define PCIFR (*(volatile unsigned char *)0x3B)
#define  PCIF3    3
#define  PCIF2    2
#define  PCIF1    1
#define  PCIF0    0
#define PCICR (*(volatile unsigned char *)0x68)
#define  PCIE3    3
#define  PCIE2    2
#define  PCIE1    1
#define  PCIE0    0
#define PCMSK0 (*(volatile unsigned char *)0x6B)
#define PCMSK1 (*(volatile unsigned char *)0x6C)
#define PCMSK2 (*(volatile unsigned char *)0x6D)
#define PCMSK3 (*(volatile unsigned char *)0x73)

/* GPIOR */
#define GPIOR0 (*(volatile unsigned char *)0x3E)
#define GPIOR1 (*(volatile unsigned char *)0x4A)
#define GPIOR2 (*(volatile unsigned char *)0x4B)

/* EEPROM */
#define EECR (*(volatile unsigned char *)0x3F)
#define  EEPM1    5
#define  EEPM0    4
#define  EERIE    3
```

```
#define  EEMPE    2
#define  EEMWE    2
#define  EEPE     1
#define  EEWE     1
#define  EERE     0
#define EEDR (*(volatile unsigned char *)0x40)
#define EEAR (*(volatile unsigned int *)0x41)
#define EEARL (*(volatile unsigned char *)0x41)
#define EEARH (*(volatile unsigned char *)0x42)

/* GTCCR */
#define GTCCR (*(volatile unsigned char *)0x43)
#define  TSM      7
#define  PSRASY   1
#define  PSR2     1
#define  PSRSYNC  0
#define  PSR10    0

/* Timer/Counter 0 */
#define OCR0B (*(volatile unsigned char *)0x48)
#define OCR0A (*(volatile unsigned char *)0x47)
#define TCNT0 (*(volatile unsigned char *)0x46)
#define TCCR0B (*(volatile unsigned char *)0x45)
#define  FOC0A    7
#define  FOC0B    6
#define  WGM02    3
#define  CS02     2
#define  CS01     1
#define  CS00     0
#define TCCR0A (*(volatile unsigned char *)0x44)
#define  COM0A1   7
#define  COM0A0   6
#define  COM0B1   5
#define  COM0B0   4
#define  WGM01    1
#define  WGM00    0

/* SPI */
#define SPCR (*(volatile unsigned char *)0x4C)
```

```
#define  SPIE     7
#define  SPE      6
#define  DORD     5
#define  MSTR     4
#define  CPOL     3
#define  CPHA     2
#define  SPR1     1
#define  SPR0     0
#define SPSR (*(volatile unsigned char *)0x4D)
#define  SPIF     7
#define  WCOL     6
#define  SPI2X    0
#define SPDR (*(volatile unsigned char *)0x4E)

/* Analog Comparator Control and Status Register */
#define ACSR (*(volatile unsigned char *)0x50)
#define  ACD      7
#define  ACBG     6
#define  ACO      5
#define  ACI      4
#define  ACIE     3
#define  ACIC     2
#define  ACIS1    1
#define  ACIS0    0

/* OCDR */
#define OCDR (*(volatile unsigned char *)0x51)
#define  IDRD     7

/* MCU */
#define MCUSR (*(volatile unsigned char *)0x54)
#define  JTRF     4
#define  WDRF     3
#define  BORF     2
#define  EXTRF    1
#define  PORF     0
#define MCUCR (*(volatile unsigned char *)0x55)
#define  JTD      7
#define  PUD      4
```

```
#define  IVSEL    1
#define  IVCE     0

#define SMCR (*(volatile unsigned char *)0x53)
#define  SM2      3
#define  SM1      2
#define  SM0      1
#define  SE       0

/* SPM Control and Status Register */
#define SPMCSR (*(volatile unsigned char *)0x57)
#define  SPMIE    7
#define  RWWSB    6
#define  SIGRD    5
#define  RWWSRE   4
#define  BLBSET   3
#define  PGWRT    2
#define  PGERS    1
#define  SPMEN    0

/* Stack Pointer */
#define SP   (*(volatile unsigned int *)0x5D)
#define SPL  (*(volatile unsigned char *)0x5D)
#define SPH  (*(volatile unsigned char *)0x5E)

/* Status REGister */
#define SREG (*(volatile unsigned char *)0x5F)

/* Watchdog Timer Control Register */
#define WDTCSR (*(volatile unsigned char *)0x60)
#define WDTCR (*(volatile unsigned char *)0x60)
#define  WDIF     7
#define  WDIE     6
#define  WDP3     5
#define  WDCE     4
#define  WDE      3
#define  WDP2     2
#define  WDP1     1
#define  WDP0     0
```

```
/* clock prescaler control register */
#define CLKPR (*(volatile unsigned char *)0x61)
#define  CLKPCE   7
#define  CLKPS3   3
#define  CLKPS2   2
#define  CLKPS1   1
#define  CLKPS0   0

/* PRR */
#define PRR0 (*(volatile unsigned char *)0x64)
#define  PRTWI    7
#define  PRTIM2   6
#define  PRTIM0   5
#define  PRUSART1 4
#define  PRTIM1   3
#define  PRSPI    2
#define  PRUSART0 1
#define  PRADC    0

/* Oscillator Calibration Register */
#define OSCCAL (*(volatile unsigned char *)0x66)

/* ADC */
#define ADC  (*(volatile unsigned int *)0x78)
#define ADCL (*(volatile unsigned char *)0x78)
#define ADCH (*(volatile unsigned char *)0x79)
#define ADCSRA (*(volatile unsigned char *)0x7A)
#define  ADEN     7
#define  ADSC     6
#define  ADATE    5
#define  ADIF     4
#define  ADIE     3
#define  ADPS2    2
#define  ADPS1    1
#define  ADPS0    0
#define ADCSRB (*(volatile unsigned char *)0x7B)
#define  ACME     6
#define  ADTS2    2
```

```
#define  ADTS1    1
#define  ADTS0    0
#define ADMUX (*(volatile unsigned char *)0x7C)
#define  REFS1    7
#define  REFS0    6
#define  ADLAR    5
#define  MUX4     4
#define  MUX3     3
#define  MUX2     2
#define  MUX1     1
#define  MUX0     0

/* DIDR */
#define DIDR0 (*(volatile unsigned char *)0x7E)
#define  ADC7D    7
#define  ADC6D    6
#define  ADC5D    5
#define  ADC4D    4
#define  ADC3D    3
#define  ADC2D    2
#define  ADC1D    1
#define  ADC0D    0
#define DIDR1 (*(volatile unsigned char *)0x7F)
#define  AIN1D    1
#define  AIN0D    0

/* Timer/Counter1 */
#define ICR1 (*(volatile unsigned int *)0x86)
#define ICR1L (*(volatile unsigned char *)0x86)
#define ICR1H (*(volatile unsigned char *)0x87)
#define OCR1B (*(volatile unsigned int *)0x8A)
#define OCR1BL (*(volatile unsigned char *)0x8A)
#define OCR1BH (*(volatile unsigned char *)0x8B)
#define OCR1A (*(volatile unsigned int *)0x88)
#define OCR1AL (*(volatile unsigned char *)0x88)
#define OCR1AH (*(volatile unsigned char *)0x89)
#define TCNT1 (*(volatile unsigned int *)0x84)
#define TCNT1L (*(volatile unsigned char *)0x84)
#define TCNT1H (*(volatile unsigned char *)0x85)
```

```
#define TCCR1C (*(volatile unsigned char *)0x82)
#define  FOC1A    7
#define  FOC1B    6
#define TCCR1B (*(volatile unsigned char *)0x81)
#define  ICNC1    7
#define  ICES1    6
#define  WGM13    4
#define  WGM12    3
#define  CS12     2
#define  CS11     1
#define  CS10     0
#define TCCR1A (*(volatile unsigned char *)0x80)
#define  COM1A1   7
#define  COM1A0   6
#define  COM1B1   5
#define  COM1B0   4
#define  WGM11    1
#define  WGM10    0

/* Timer/Counter2 */
#define ASSR (*(volatile unsigned char *)0xB6)
#define  EXCLK    6
#define  AS2      5
#define  TCN2UB   4
#define  OCR2AUB  3
#define  OCR2BUB  2
#define  TCR2AUB  1
#define  TCR2BUB  0
#define OCR2B (*(volatile unsigned char *)0xB4)
#define OCR2A (*(volatile unsigned char *)0xB3)
#define TCNT2 (*(volatile unsigned char *)0xB2)
#define TCCR2B (*(volatile unsigned char *)0xB1)
#define  FOC2A    7
#define  FOC2B    6
#define  WGM22    3
#define  CS22     2
#define  CS21     1
#define  CS20     0
#define TCCR2A (*(volatile unsigned char *)0xB0)
```

```
#define  COM2A1   7
#define  COM2A0   6
#define  COM2B1   5
#define  COM2B0   4
#define  WGM21    1
#define  WGM20    0

/* 2-wire SI */
#define TWBR (*(volatile unsigned char *)0xB8)
#define TWSR (*(volatile unsigned char *)0xB9)
#define  TWPS1    1
#define  TWPS0    0
#define TWAR (*(volatile unsigned char *)0xBA)
#define  TWGCE    0
#define TWDR (*(volatile unsigned char *)0xBB)
#define TWCR (*(volatile unsigned char *)0xBC)
#define  TWINT    7
#define  TWEA     6
#define  TWSTA    5
#define  TWSTO    4
#define  TWWC     3
#define  TWEN     2
#define  TWIE     0
#define TWAMR (*(volatile unsigned char *)0xBD)

/* USART0 */
#define UBRR0H (*(volatile unsigned char *)0xC5)
#define UBRR0L (*(volatile unsigned char *)0xC4)
#define UBRR0 (*(volatile unsigned int *)0xC4)
#define UCSR0C (*(volatile unsigned char *)0xC2)
#define  UMSEL01  7
#define  UMSEL00  6
#define  UPM01    5
#define  UPM00    4
#define  USBS0    3
#define  UCSZ01   2
#define  UCSZ00   1
#define  UCPOL0   0
#define UCSR0B (*(volatile unsigned char *)0xC1)
```

```
#define  RXCIE0   7
#define  TXCIE0   6
#define  UDRIE0   5
#define  RXEN0    4
#define  TXEN0    3
#define  UCSZ02   2
#define  RXB80    1
#define  TXB80    0
#define UCSR0A (*(volatile unsigned char *)0xC0)
#define  RXC0     7
#define  TXC0     6
#define  UDRE0    5
#define  FE0      4
#define  DOR0     3
#define  UPE0     2
#define  U2X0     1
#define  MPCM0    0
#define UDR0 (*(volatile unsigned char *)0xC6)

/* USART1 */
#define UBRR1H (*(volatile unsigned char *)0xCD)
#define UBRR1L (*(volatile unsigned char *)0xCC)
#define UBRR1 (*(volatile unsigned int *)0xCC)
#define UCSR1C (*(volatile unsigned char *)0xCA)
#define  UMSEL11  7
#define  UMSEL10  6
#define  UPM11    5
#define  UPM10    4
#define  USBS1    3
#define  UCSZ11   2
#define  UCSZ10   1
#define  UCPOL1   0
#define UCSR1B (*(volatile unsigned char *)0xC9)
#define  RXCIE1   7
#define  TXCIE1   6
#define  UDRIE1   5
#define  RXEN1    4
#define  TXEN1    3
#define  UCSZ12   2
```

```
#define  RXB81    1
#define  TXB81    0
#define UCSR1A (*(volatile unsigned char *)0xC8)
#define  RXC1     7
#define  TXC1     6
#define  UDRE1    5
#define  FE1      4
#define  DOR1     3
#define  UPE1     2
#define  U2X1     1
#define  MPCM1    0
#define UDR1 (*(volatile unsigned char *)0xCE)


/* bits */

/* Port A */
#define  PORTA7   7
#define  PORTA6   6
#define  PORTA5   5
#define  PORTA4   4
#define  PORTA3   3
#define  PORTA2   2
#define  PORTA1   1
#define  PORTA0   0
#define  PA7      7
#define  PA6      6
#define  PA5      5
#define  PA4      4
#define  PA3      3
#define  PA2      2
#define  PA1      1
#define  PA0      0
#define  DDA7     7
#define  DDA6     6
#define  DDA5     5
#define  DDA4     4
#define  DDA3     3
#define  DDA2     2
```

```
#define  DDA1     1
#define  DDA0     0
#define  PINA7    7
#define  PINA6    6
#define  PINA5    5
#define  PINA4    4
#define  PINA3    3
#define  PINA2    2
#define  PINA1    1
#define  PINA0    0

/* Port B */
#define  PORTB7   7
#define  PORTB6   6
#define  PORTB5   5
#define  PORTB4   4
#define  PORTB3   3
#define  PORTB2   2
#define  PORTB1   1
#define  PORTB0   0
#define  PB7      7
#define  PB6      6
#define  PB5      5
#define  PB4      4
#define  PB3      3
#define  PB2      2
#define  PB1      1
#define  PB0      0
#define  DDB7     7
#define  DDB6     6
#define  DDB5     5
#define  DDB4     4
#define  DDB3     3
#define  DDB2     2
#define  DDB1     1
#define  DDB0     0
#define  PINB7    7
#define  PINB6    6
#define  PINB5    5
```

```
#define  PINB4    4
#define  PINB3    3
#define  PINB2    2
#define  PINB1    1
#define  PINB0    0

/* Port C */
#define  PORTC7   7
#define  PORTC6   6
#define  PORTC5   5
#define  PORTC4   4
#define  PORTC3   3
#define  PORTC2   2
#define  PORTC1   1
#define  PORTC0   0
#define  PC7      7
#define  PC6      6
#define  PC5      5
#define  PC4      4
#define  PC3      3
#define  PC2      2
#define  PC1      1
#define  PC0      0
#define  DDC7     7
#define  DDC6     6
#define  DDC5     5
#define  DDC4     4
#define  DDC3     3
#define  DDC2     2
#define  DDC1     1
#define  DDC0     0
#define  PINC7    7
#define  PINC6    6
#define  PINC5    5
#define  PINC4    4
#define  PINC3    3
#define  PINC2    2
#define  PINC1    1
#define  PINC0    0
```

```
/* Port D */
#define  PORTD7   7
#define  PORTD6   6
#define  PORTD5   5
#define  PORTD4   4
#define  PORTD3   3
#define  PORTD2   2
#define  PORTD1   1
#define  PORTD0   0
#define  PD7      7
#define  PD6      6
#define  PD5      5
#define  PD4      4
#define  PD3      3
#define  PD2      2
#define  PD1      1
#define  PD0      0
#define  DDD7     7
#define  DDD6     6
#define  DDD5     5
#define  DDD4     4
#define  DDD3     3
#define  DDD2     2
#define  DDD1     1
#define  DDD0     0
#define  PIND7    7
#define  PIND6    6
#define  PIND5    5
#define  PIND4    4
#define  PIND3    3
#define  PIND2    2
#define  PIND1    1
#define  PIND0    0

/* PCMSK3 */
#define  PCINT31  7
#define  PCINT30  6
#define  PCINT29  5
```

```
#define  PCINT28  4
#define  PCINT27  3
#define  PCINT26  2
#define  PCINT25  1
#define  PCINT24  0
/* PCMSK2 */
#define  PCINT23  7
#define  PCINT22  6
#define  PCINT21  5
#define  PCINT20  4
#define  PCINT19  3
#define  PCINT18  2
#define  PCINT17  1
#define  PCINT16  0
/* PCMSK1 */
#define  PCINT15  7
#define  PCINT14  6
#define  PCINT13  5
#define  PCINT12  4
#define  PCINT11  3
#define  PCINT10  2
#define  PCINT9   1
#define  PCINT8   0
/* PCMSK0 */
#define  PCINT7   7
#define  PCINT6   6
#define  PCINT5   5
#define  PCINT4   4
#define  PCINT3   3
#define  PCINT2   2
#define  PCINT1   1
#define  PCINT0   0


/* Lock and Fuse Bits with LPM/SPM instructions */

/* lock bits */
#define  BLB12    5
#define  BLB11    4
```

```
#define  BLB02    3
#define  BLB01    2
#define  LB2      1
#define  LB1      0

/* fuses low bits */
#define  CKDIV8   7
#define  CKOUT    6
#define  SUT1     5
#define  SUT0     4
#define  CKSEL3   3
#define  CKSEL2   2
#define  CKSEL1   1
#define  CKSEL0   0

/* fuses high bits */
#define  OCDEN    7
#define  JTAGEN   6
#define  SPIEN    5
#define  WDTON    4
#define  EESAVE   3
#define  BOOTSZ1  2
#define  BOOTSZ0  1
#define  BOOTRST  0

/* extended fuses */
#define  BODLEVEL2 2
#define  BODLEVEL1 1
#define  BODLEVEL0 0


/* Interrupt Vector Numbers */

#define iv_RESET        1
#define iv_INT0         2
#define iv_EXT_INT0     2
#define iv_INT1         3
#define iv_EXT_INT1     3
#define iv_INT2         4
```

```
#define iv_EXT_INT2     4
#define iv_PCINT0       5
#define iv_PCINT1       6
#define iv_PCINT2       7
#define iv_PCINT3       8
#define iv_WDT          9
#define iv_TIMER2_COMPA 10
#define iv_TIMER2_COMPB 11
#define iv_TIMER2_OVF   12
#define iv_TIM2_COMPA   10
#define iv_TIM2_COMPB   11
#define iv_TIM2_OVF     12
#define iv_TIMER1_CAPT  13
#define iv_TIMER1_COMPA 14
#define iv_TIMER1_COMPB 15
#define iv_TIMER1_OVF   16
#define iv_TIM1_CAPT    13
#define iv_TIM1_COMPA   14
#define iv_TIM1_COMPB   15
#define iv_TIM1_OVF     16
#define iv_TIMER0_COMPA 17
#define iv_TIMER0_COMPB 18
#define iv_TIMER0_OVF   19
#define iv_TIM0_COMPA   17
#define iv_TIM0_COMPB   18
#define iv_TIM0_OVF     19
#define iv_SPI_STC      20
#define iv_USART0_RX    21
#define iv_USART0_RXC   21
#define iv_USART0_DRE   22
#define iv_USART0_UDRE  22
#define iv_USART0_TX    23
#define iv_USART0_TXC   23
#define iv_ANA_COMP     24
#define iv_ANALOG_COMP  24
#define iv_ADC          25
#define iv_EE_RDY       26
#define iv_EE_READY     26
#define iv_TWI          27
```

```
#define iv_TWSI         27
#define iv_SPM_RDY      28
#define iv_SPM_READY    28
#define iv_USART1_RX    29
#define iv_USART1_RXC   29
#define iv_USART1_DRE   30
#define iv_USART1_UDRE  30
#define iv_USART1_TX    31
#define iv_USART1_TXC   31

/* */

#endif
```

Author's Biography

STEVEN F. BARRETT

Steven F. Barrett, Ph.D., P.E., received the BS Electronic Engineering Technology from the University of Nebraska at Omaha in 1979, the M.E.E.E. from the University of Idaho at Moscow in 1986, and the Ph.D. from The University of Texas at Austin in 1993. He was formally an active duty faculty member at the United States Air Force Academy, Colorado and is now the Associate Dean of Academic Programs at the University of Wyoming. He is a member of IEEE (senior) and Tau Beta Pi (chief faculty advisor). His research interests include digital and analog image processing, computer-assisted laser surgery, and embedded controller systems. He is a registered Professional Engineer in Wyoming and Colorado. He co-wrote with Dr. Daniel Pack six 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