Tengis Tserendondog

CONTROL TECHNIQUE WITH MICROCONTROLLERS

Co-funded by the
Erasmus+ Programme
of the European Union

MONGOLIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CONTROL TECHNIQUE WITH MICROCONTROLLERS

Tengis Tserendondog

2020

Contents

CHAPTER 1

Introduction

Topic 1

Introduction

- Handbook organization
- Brief introductions of chapters
- Control technology
- Microcontroller

Today, the most popular and powerful control technologies applied in our routine life are automatic and intelligent controls. In this book, we tried to provide a knowledge of classical control technologies with a real microcontroller system AVR Atmega32.
It always felt that there is a gap that existed between the control theories and practical control implementations, and this gap is not easy to be covered by most of the current control books, including textbooks. The objective of this handbook is to provide students a way or a bridge to cross over the gap mentioned above to enable them to combine control theories they learned from classes with popular practical control targets together to actually design, build, simulate, and finally develop real control programs to perform practical real-time closed-loop control strategies to real control targets to get optimal control systems.
Students need to get much more detailed and actual control knowledge as well as techniques for most popular and practical control systems, and furthermore to utilize the knowledge and techniques, they learned from classes to personally and practically design a complete control system with actual control objectives or targets to fully understand what they have learned. For that purpose, a real and actual controller or a microcontroller system is needed as a tool to enable students to develop control algorithms and programs on it to realize the actual control functions for control targets or plants.

This handbook is about the theory and practice of microcontroller-based automatic control systems engineering. With the help of these hardware and software as well as practical application notes with real examples, students can design, develop, and build some real and actual control systems by developing practical programming codes to control microcontrollers to perform real-time controls to some motor systems. All example projects in the book have been compiled, built, and tested to help students to master the main techniques and ideas.

This book is composed of nine chapters with an easy study way to enable students to learn classical control technologies effectively and practically. Each chapter contains home works and exercises as well as lab projects to enable students to perform necessary exercises to improve their learning and understanding of the related materials and technologies.

Chapter 1: Provide an overview and introduction about the book with highlights of outstanding features and organizations of the book.

Chapter 2: Fundamental and basic control technologies, including the classic control strategy, is discussed and introduced in this chapter. Both the open-loop and closed-loop control systems are introduced with some actual examples.

Chapter 3: Sensors are an important part of closed-loop systems. In this chapter, we will introduce analog and digital sensors and its important factors. This chapter discusses commonly used sensors such as temperature, position, force, velocity and acceleration sensor.

Chapter 4: Electric motors are the most prevalent "machines" in use in process plants. This chapter first introduces the reader to this machine category, followed by basic motor types. In this chapter, we will study the basic principle of operation and their characteristics. It's important to understand motor characteristics so we can choose the right one for our application requirements. We present two kinds of motors, DC motor, and Stepper motor. We will learn how to control the speed of these motors.

Chapter 5: In this chapter explain the role of the CPU, memory and I/O device in a microprocessor system. Distinguish between the microprocessor and microcontroller.

Chapter 6: The handbook is based on the C programming language. Chapter 6 gives a brief introduction to the features of this language.

Chapter 7: This chapter is devoted to the popular AVR microcontroller family which is described and used in this book. This chapter is a general introduction to AVR microcontrollers regarding their features and capabilities. The AVR features such as the internal architecture, the memory system, and the registers are presented in this chapter. These features are presented only with the needed level of details in order to be easily understandable by the reader without any hard work.

Chapter 8:: The main topic of this chapter is concentrated on the PID control system design and analysis. The stability of a control system is one of the most important topics in control engineering.

Chapter 9: Presents a case study. Programming examples are given to show how a particular realization can be programmed and implemented on a microcontroller.

CHAPTER 2

Control System

Topic 2

Control System

- Open loop system
- Closed loop system
- Feedback control system
- Transfer function

System is a group of related things that work together as a whole[1]. These things can be real or imaginary. Systems can be man-made things like a car engine or natural things like a star system. Systems can also be concepts made by people to organize ideas. A **control system** manages, commands, directs, or regulates the behavior of other devices or systems using control loops [2]. It can range from a single home heating controller using a thermostat controlling a domestic boiler to large industrial control systems that are used for controlling processes or machines.
Two popular control systems are **open-loop** and **closed-loop** control systems[1]. A control system should contain a controller to perform various control functions to the system to get the desired outputs. Therefore, a complete control system should be composed of inputs, the controller, the system or a process, and the system outputs. Most control systems in our world are closed-loop control systems, but some of them belong to open-loop control systems.

2.1 Open loop system

In an open-loop control system, the control action from the controller is independent of the process variable. An example of this is a central heating boiler controlled only by a timer. The control action is switching on or off of the boiler. The process variable is the building temperature. This controller operates the heating system for a constant time regardless of the temperature of the building.
Another open-loop control system is a microwave oven and its operational procedure can be illustrated in Fig 2.1.

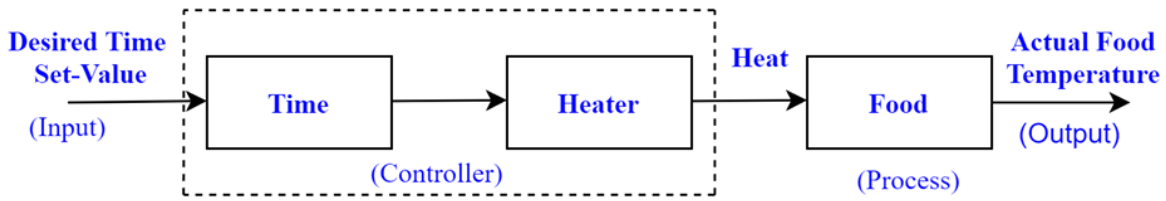

FIGURE 2.1: Open loop system

By setting the desired time interval (set value), which can be considered as the input, the timer, and the heater (controller) can start and delay the desired period of time to enable the heater to heat the food (process) to the appropriate temperature (output).
The reason we call this kind of control system as an open-loop system is that the output has no influence or effect on the control action of the input signal. In other words, in this open-loop control system, the output is neither measured nor feedback for comparison with the input. Therefore, an open-loop system is expected to faithfully follow its input command or set point regardless of the final result. This control system is also called a **non-feedback control system** [1]. Furthermore, an open-loop system has no knowledge about the output condition, such as the food temperature, so it cannot self-correct any errors it could make when the set value drifts, even if this results in large deviations from the set value. Another disadvantage of an open-loop system is that they cannot handle disturbances or changes in the conditions which may reduce its ability to complete the desired task.
For example, the microwave door opens and the heat is lost. The timing controller continues regardless of the full-time interval but the food is not heated at the end of the heating process. This is because there is no information feedback to maintain a constant temperature that is equal or closed to the input (set value). One of the possible problems is that the open-loop system errors, such as environmental temperature changing or timer fault operations, can disturb the food heating process and, therefore, requires extra supervisory attention of a user such as an operator. The problem with this anticipatory control approach is that the user needs to monitor the process temperature frequently and take any corrective control action whenever the food heating process deviates from its desired value of the set value. This kind of disturbance could be reduced by periodically monitoring the relationship between the set value and the motor running speed, and by increasing or decreasing the set value manually.
The advantages of this open-loop control are that it is low-cost and simple and easy to implement, thus making it ideal for use in well defined systems where the relationship between the input and the output are direct and not influenced by outside disturbances.

However, open-loop control is useful and economic for well defined systems where the

relationship between input and the output can be reliably modeled by a mathematical formula. For example, determining the voltage to be fed to an electric DC motor that drives a constant load, in order to achieve a desired rotating speed would be a good application. But if the loads were not predictable and became excessive, the motor's speed might vary as a function of the loads and not just the input voltage and an open-loop controller would be insufficient to ensure repeatable control of the velocity.

Need to remember

Based on the above discussions, an open-loop control system has the following potential problems:

- There is no comparison between actual input and desired output values.
- Has no self-regulation or self control action over the output value.
- Each input settings determines a fixed output value for the controller.
- Cannot reduce or overcome all variations or disturbances coming from the external conditions.

In order to solve these potential problems and improve the performance of the entire control system, we need to take a look at another type of control system, closed-loop, or feedback control system.

2.2 Closed loop System

A closed-loop control system that is also known as a **feedback control system** is a control system that uses the concept of an open-loop system as its forward path has one or more feedback loops or paths between its output and its input[1]. The reference to feedback simply means that some portion of the output is returned back to the input to form part of the system's excitation.

Important

Closed-loop systems are designed to automatically achieve and maintain the desired output by comparing it with the actual input condition.

It does this by comparing a part of the entire feedback from the output and the desired input to generate an error signal, which is the difference between the output and the reference input.

$$Error = Input - Output$$

In other words, a closed-loop system is a fully automatic control system in which its control action being dependent on the output in some way.
Our previous microwave oven as an example, and it can become a closed-loop control system by adding a feedback path with a sensor that is used to detect the actual temperature of the food and a comparator that is used to compare the feedback output with the input (set value). An error signal can be obtained by comparing the feedback output and the input, and this error signal can be used as input to the controller to automatically adjust the output to make it to the desired input set value or make the output as closely equal to the input as possible. This sensor would monitor the actual temperature of the food and compare it with or subtract it from the input reference. The error signal is then amplified by the controller, and the controller output makes the necessary correction to the heating system to reduce any error.
For example, if the food is not hot enough, the controller may increase the temperature or the heating time. Likewise, if the food is very hot, then the controller may reduce the temperature or stop the process so as not to overheat or burn the food.
The closed-loop configuration is defined by the feedback signal, derived from the sensor or a thermometer in our food heating system.
The magnitude and polarity of the error signal would be directly related to the difference between the required heating temperature and actual food temperature. The term closed-loop control always means the use of feedback control in order to reduce any errors between the output and the input. Just because of this feedback, it distinguishes the major differences between an open-loop and a closed-loop system.
The accuracy of the output thus depends on the feedback path, which in general can be made very accurate and within electronic control systems and circuits.
From this simple example, it can be found that a closed-loop system has many advantages over open-loop systems. The primary advantage of a closed-loop feedback control system is its ability to reduce a system's sensitivity to external disturbances and can automatically correct or modify the error to make the output as equal to the input as possible.
Another point to be noted for this feedback control system is that the feedback path or the sensor must be able to provide the two following functions:

1. Direct a part of the entire output back to the input to allow the comparator to compare the input and the output to get a difference or an error signal as an updated input to the controller.

2. Convert the output type to the input type to make this comparison possible.

As shown in our food heating closed-loop control system in Fig.2.2, the sensor not only provides feedback from the output to the input but also needs to convert the output type (food temperature) to the input type (timer setup value). By adjusting the timer setup

value, the food temperature can be modified and changed to the desired temperature. Otherwise, the comparison between the input and the output cannot be executed because of the different types of input and output.

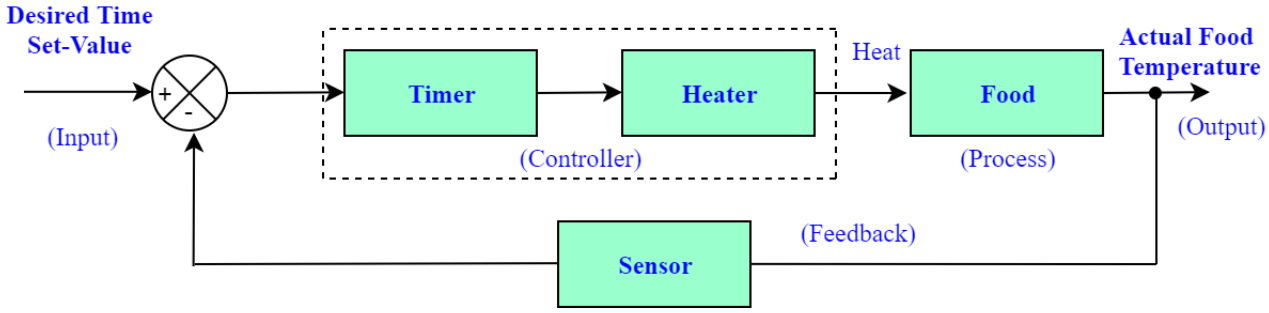

FIGURE 2.2: Closed loop system

Compared to the open-loop control system, closed-loop control systems have many advantages over open-loop systems. One advantage is that the use of feedback makes the system response relatively insensitive to external disturbances and internal variations in system parameters such as temperature or variation on elements. It is thus possible to use relatively inaccurate and cheaper components to obtain the accurate control of a given process or plant.

Need to remember

Based on the discussions above, we can conclude that a closed-loop control system is better than an open-loop control system with the following advantages:

1. Reducing errors by automatically adjusting the system's input.
2. Improving the stability of an unstable system.
3. Reducing the system's sensitivity to enhance robustness against external disturbances or internal variations to the process.
4. Producing a reliable and repeatable performance.

2.3 System Design

A control system consists of **subsystems** and **processes** (or plants) assembled for the purpose of obtaining the desired output with desired performance, given a specified input. Figure 2.3 shows a control system in its simplest form, where the input represents the desired output.

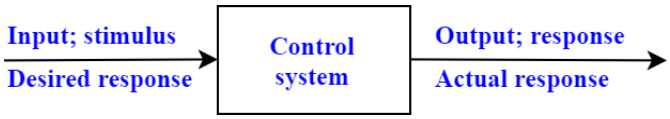

FIGURE 2.3: Control system

For example, consider an elevator. The push of the fourth-floor button is an input that represents our desired output, shown as a step function in Figure 2.4. The performance of the elevator can be seen from the elevator response curve in the figure.

Important

Two major measures of performance are apparent:
(1) **the transient response**
(2) **the steady-state error**

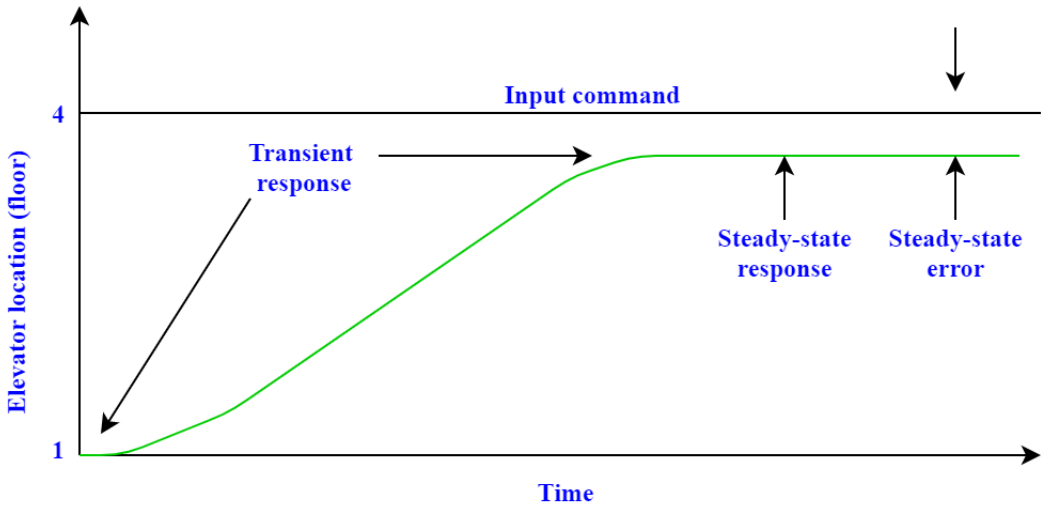

FIGURE 2.4: Transient response

In our example, passenger comfort and passenger patience are dependent upon the transient response. If this response is too fast, passenger comfort is sacrificed; if too slow, passenger patience is sacrificed. The steady-state error is another important performance

specification since passenger safety and convenience would be sacrificed if the elevator did not properly level.

We briefly allude to some control system performance specifications, such as transient response and steady-state error.

The **analysis** is the process by which a system's performance is determined. For example, we evaluate its transient response and steady-state error to determine if they meet the desired specifications.

Design is the process by which a system's performance is created or changed. For example, if a system's transient response and steady-state error are analyzed and found not to meet the specifications, then we change parameters or add additional components to meet the specifications.

Transient response is important. In the case of an elevator, a slow transient response makes passengers impatient, whereas an excessively rapid response makes them uncomfortable. If the elevator oscillates about the arrival floor for more than a second, a disconcerting feeling can result. Transient response is also important for structural reasons: Too fast a transient response could cause permanent physical damage.

Another analysis and design goal focuses on the steady-state response. As we have seen, this response resembles the input and is usually what remains after the transients have decayed to zero. We are concerned about the accuracy of the steady-state response. We define steady-state errors quantitatively, analyze a system's steady-state error, and then design corrective action to reduce the steady-state error, our second analysis and design objective.

Discussion of transient response and steady-state error is moot if the system does not have stability. Control systems must be designed to be **stable**. That is, their natural response must decay to zero as the time approaches infinity, or oscillate. In many systems, the transient response you see on a time response plot can be directly related to the natural response. Thus, if the natural response decays to zero as the time approaches infinity, the transient response will also die out, leaving only the forced response. If the system is stable, the proper transient response and steady-state error characteristics can be designed. Stability is our third analysis and design objective.

2.4 Control System Transfer Function

A transfer function represents the relationship between the output signal of a control system and the input signal, for all possible input values. A block diagram is a visualization of the control system which uses blocks to represent the transfer function, and arrows which represent the various input and output signals.

FIGURE 2.5: Transfer function

In a Laplace Transform, if the input is represented by R(s) and the output is represented by C(s), then the transfer function will be:

$$G(s) = \frac{C(s)}{R(s)} \tag{2.1}$$

$$R(s)G(s) = C(s) \tag{2.2}$$

It is not necessary that the output and input of a control system are of the same category. For example, in electric motors, the input is an electrical signal whereas the output is mechanical signal since electrical energy required to rotate the motors. Similarly in an electric generator, the input is a mechanical signal and the output is an electrical signal since mechanical energy is required to produce electricity in a generator. But for mathematical analysis, of a system, all kinds of signals should be represented in a similar form. This is done by transforming all kinds of signals to their Laplace form. Also, the transfer function of a system is represented by Laplace form by dividing output Laplace transfer function to input Laplace transfer function. Hence a basic block diagram of a control system can be represented as

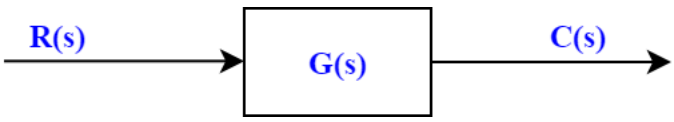

FIGURE 2.6: Transfer function

There are major two ways of obtaining a transfer function for the control system. The ways are:

• **Block Diagram Method:** It is not convenient to derive a complete transfer function for a complex control system. Therefore the transfer function of each element of a control

system is represented by a block diagram. Block diagram reduction techniques are applied to obtain the desired transfer function.

• **Signal Flow Graphs:** The modified form of a block diagram is a signal flow graph. The block diagram gives a pictorial representation of a control system. The signal flow graph further shortens the representation of a control system.

Review questions

1. What is the open-loop control system? Draw its block diagram
2. What is the closed-loop control system? Draw its block diagram
3. What is the major difference between the open-loop control system and closed-loop control system?
4. Indicate which are open-loop or closed-loop control systems for the following systems.

a. The man driving car
b. Traffic light
c. white line following robot
d. Toaster
e. A microwave heating process.

5. Draw a block diagram of a home heating system, and identify the function of each component and input and output signals.
6. What is the transient response?
7. What is the steady-state error?
8. Draw the transient response of the elevator that goes very fast.
9. What is the system transfer function?
10. Draw a block diagram of a toaster, imagine if it's a closed-loop system.

CHAPTER 3

Sensors used in Microcontroller

Topic 3

Sensors

- Analog vs Digital sensors
- Sensor's factors
- Sensor's error
- Sensor types

Sensor - an electrical/mechanical/chemical device that maps an environmental attribute to a quantitative measurement [3]. Each sensor is based on a transduction principle - conversion of energy from one form to another.

Need to remember

Classification of Sensors:

- Internal state vs. external state
  - feedback of robot internal parameters, e.g. battery level, wheel position, joint angle, etc,
  - observation of environments, objects
- Active vs. non-active
  - emitting energy into the environment, e.g., radar, sonar
  - passively receive energy to make observation, e.g., camera
- Contact vs. non-contact
- Visual vs. non-visual
  - vision-based sensing, image processing, video camera

Sensors are an important part of closed-loop systems [4]. A sensor is a device that outputs a signal which is related to the measurement of a physical quantity such as temperature, speed, force, pressure, displacement, acceleration, torque, flow, light or sound. Sensors are used in closed-loop systems in the feedback loops, and they provide information about the actual output of a plant. For example, a speed sensor gives a signal proportional to the speed of a motor and this signal is subtracted from the desired speed reference input in order to obtain the error signal.

Sensors can be classified as analog or digital. **Analog sensors** are more widely available, and their outputs are analog voltages. For example, the output of an analog temperature sensor may be a voltage proportional to the measured temperature. Analog sensors

can only be connected to a computer by using an A/D converter. **Digital sensors** are not very common and they have logic level outputs that can directly be connected to a computer input port. The choice of a sensor for a particular application depends on many factors such as the cost, reliability, required accuracy, resolution, range and linearity of the sensor.

Important

Some important factors are described below.

Range. The range of a sensor specifies the upper and lower limits of the measured variable for which a measurement can be made. For example, if the range of a temperature sensor is specified as 10–60°C then the sensor should only be used to measure temperatures within that range.

Resolution. The resolution of a sensor is specified as the largest change in measured value that will not result in a change in the sensor's output, i.e. the measured value can change by the amount quoted by the resolution before this change can be detected by the sensor. In general, the smaller this amount the better the sensor is, and sensors with a wide range have less resolution. For example, a temperature sensor with a resolution of 0.001K is better than a sensor with a resolution of 0.1K.

Repeatability. The repeatability of a sensor is the variation of output values that can be expected when the sensor measures the same physical quantity several times. For example, if the voltage across a resistor is measured at the same time several times we may get slightly different results.

Linearity. An ideal sensor is expected to have a linear transfer function, i.e. the sensor output is expected to be exactly proportional to the measured value. However, in practice, all sensors exhibit some amount of nonlinearity depending upon the manufacturing tolerances and the measurement conditions.

Dynamic response. The dynamic response of a sensor specifies the limits of the sensor characteristics when the sensor is subject to a sinusoidal frequency change. For example, the dynamic response of a microphone may be expressed in terms of the 3 dB bandwidth of its frequency response.

3.1 Temperature Sensor

Temperature is one of the fundamental physical variables in most chemical and process control applications. Accurate and reliable measurement of the temperature is important in nearly all process control applications. Temperature sensors can be analog or digital. Some of the most commonly used analog temperature sensors are thermocouples, resistance temperature detectors (RTDs) and thermistors. Digital sensors are in the form of integrated circuits. The choice of a sensor depends on the accuracy, the temperature range, speed of response, thermal coupling, the environment (chemical, electrical, or physical) and the cost.

Sensor	Temperatur range (°C)	Accuracy(±°C)	Cost	Robustness
Thermocouple	-270 to +2600	1	Low	Very high
RTD	-200 to +600	0.2	Medium	High
Thermistor	-50 to +200	0.2	Low	Medium
Integrated circuit	-40 to +125	1	Low	Low

A popular voltage output analog integrated circuit temperature sensor is the LM35, manufactured by National Semiconductors Inc. (see Figure 3.1). This is a 3-pin analog output sensor which provides a linear output voltage of 10mV/°C. The temperature range is 0°C to +100°, with an accuracy of 1.5°C.

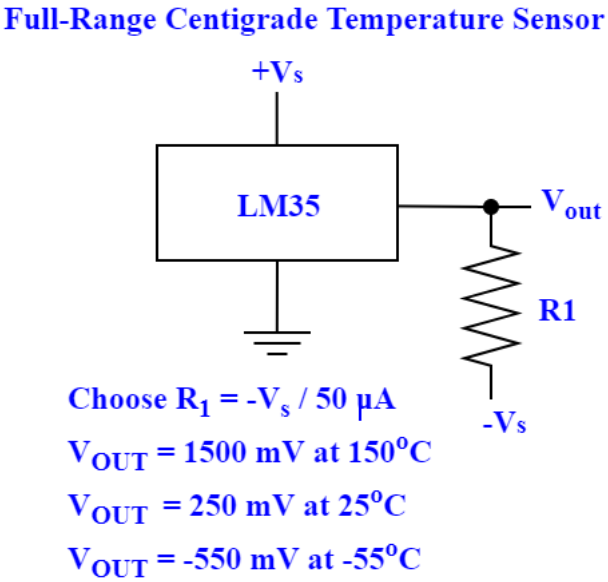

FIGURE 3.1: Tempetarure sensor

Important

There can be several sources of error during the measurement of temperature. Some important possible errors are described below.

Sensor self-heating. RTDs, thermistors and integrated circuit sensors require an external power supply for their operation. The power supply can cause the sensor to heat, leading to an error in the measurement. The effect of self-heating depends on the size of the sensor and the amount of power dissipated by the sensor. Self-heating can be avoided by using the lowest possible external power, or by considering the heating effect in the measurement.

Electrical noise. Electrical noise can introduce errors into the measurement. Thermocouples produce very low voltages and, as a result, noise can easily enter the measurement. This noise can usually be minimized by using low-pass filters, and by keeping the sensor leads as short as possible and away from motors and other electrical machinery.

Mechanical stress. Some sensors such as RTDs are sensitive to mechanical stress and should be used carefully. Mechanical stress can be minimized by avoiding the deformation of the sensor.

Thermal coupling. It is important that for accurate and fast measurements the sensor should make good contact with the measuring surface. If the surface has a thermal gradient then incorrect placement of the sensor can lead to errors. If the sensor is used in a liquid, the liquid should be stirred to cause a uniform heat distribution. Integrated circuit sensors usually suffer from thermal coupling since they are not easily mountable on surfaces.

Sensor time constant. The response time of the sensor can be another source of error. Every type of sensor takes a finite time to respond to a change in its environment. Errors due to the sensor time constant can be minimized by improving the coupling between the sensor and the measuring surface.

3.2 Position Sensor

Position sensors are used to measure the position of moving objects. These sensors are basically of two types: sensors to measure linear movement, and sensors to measure angular movement.
The most commonly used of all the “Position Sensors”, is the potentiometer because it is an inexpensive and easy to use a position sensor. Potentiometers are available in linear and rotary forms.

Potentiometer has a wiper contact linked to a mechanical shaft that can be either angular (rotational) or linear (slider type) in its movement, and which causes the resistance value between the wiper/slider and the two end connections to change giving an electrical signal output that has a proportional relationship between the actual wiper position on the resistive track and its resistance value. In other words, resistance is proportional to position.

When used as a position sensor the moveable object is connected directly to the rotational shaft or slider of the potentiometer. This configuration produces a potential or voltage divider type circuit output which is proportional to the shaft position. For example, if you apply a voltage of say 10V across the resistive element of the potentiometer the maximum output voltage would be equal to the supply voltage at 10 Volts, with the minimum output voltage equal to 0 Volts. Then the potentiometer wiper will vary the output signal from 0 to 10 Volts, with 5 Volts indicating that the wiper or slider is at its half-way or center position.

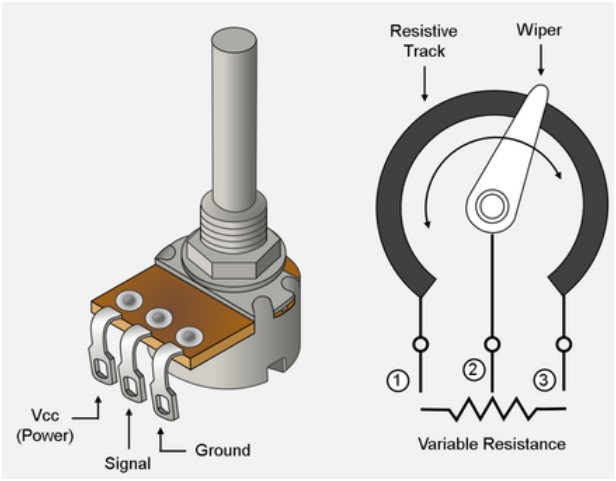

FIGURE 3.2: Potentiometer as position sensor

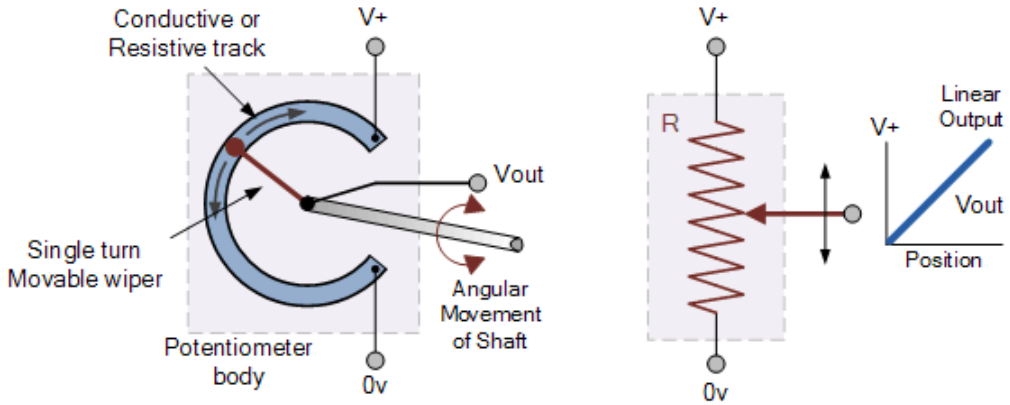

FIGURE 3.3: Potentiometer as position sensor

The rotary potentiometer can be used to measure angular position. If Vi is again the applied voltage, the voltage across the arm is given by

$$V_a = kV_i\theta \tag{3.1}$$

where θ is the angle of the arm, and k is a constant.
Potentiometer type position sensors are low-cost, but they have the disadvantage that the range is limited and also that the sensor can be worn out by the excessive movement of the shaft. Among other types of position sensors are capacitive sensors, inductive sensors, linear variable differential transformers (LVDTs) and optical encoders

3.3 Distance measuring sensors

Distance sensors generally work by outputting a signal of some kind, (eg laser, IR LED, ultrasonic waves) and then reading how it has changed on its return. That change may be in the intensity of the returned signal, the time it takes the signal to return, etc.

An ultrasonic sensor is an instrument that measures the distance to an object using ultrasonic sound waves. An ultrasonic sensor uses a transducer to send and receive ultrasonic pulses that relay back information about an object's proximity. High-frequency sound waves reflect from boundaries to produce distinct echo patterns. Ultrasonic sensors work by sending out a sound wave at a frequency above the range of human hearing. The transducer of the sensor acts as a microphone to receive and send the ultrasonic sound. The ultrasonic sensor uses a single transducer to send a pulse and to receive the echo. The sensor determines the distance to a target by measuring time lapses between the sending and receiving of the ultrasonic pulse.

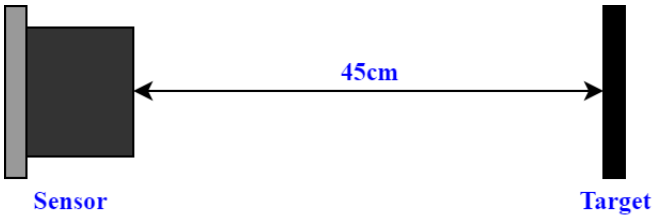

FIGURE 3.4: Ultrasonic sensor

Time of Flight: 2610 us
Speed of Sound: 58us/cm
Range(cm) = Time of Flight / Speed of Sound
Range(cm) = 2610us /(58us/cm)
Range = 45cm

The working principle of this module is simple. It sends an ultrasonic pulse out at 40 kHz which travels through the air and if there is an obstacle or object, it will bounce back to the sensor. By calculating the travel time and the speed of sound, the distance can be calculated.

A laser distance meter works by using measuring the time it takes a pulse of laser light to be reflected off a target and returned to the sender. This is known as the "time of flight" principle, and the method is known either as “time of flight” or “pulse” measurement. A laser distance meter emits a pulse of the laser at a target. The pulse then reflects off the target and back to the sending device (in this case, a laser distance meter). This "time of flight" principle is based on the fact that laser light travels at a fairly constant speed through the Earth’s atmosphere. Inside the meter, a simple computer quickly calculates the distance to a target. This method of distance calculation is capable of measuring the distance from the Earth to the moon within a few centimeters. Laser distance meters may also be referred to as “range finders” or “laser range finders.”
The distance between the meter and target is given by $D = ct/2$, where c equals the speed of light and t equals the amount of time for the round trip between meter and target. Given the high speed at which the pulse travels and its focus, this rough calculation is very accurate over distances of feet or miles but loses accuracy over much closer or farther distances.

Need to remember

Ultrasound is intrinsically less accurate because the sound is far more difficult to focus than laser light. Accuracy is typically several centimeters, compared with a few millimeters for laser. An ultrasound needs a fairly large, smooth, flat surface as the target, so that is a severe limitation. You can’t measure a narrow pipe, for example. The ultrasound signal spreads out in a cone from the meter and any objects in the way can interfere with the measurement. Even with laser aiming, you can’t always be sure that the surface from which the sound reflection is detected is the same as that where the laser dot is showing. This can lead to gross errors.

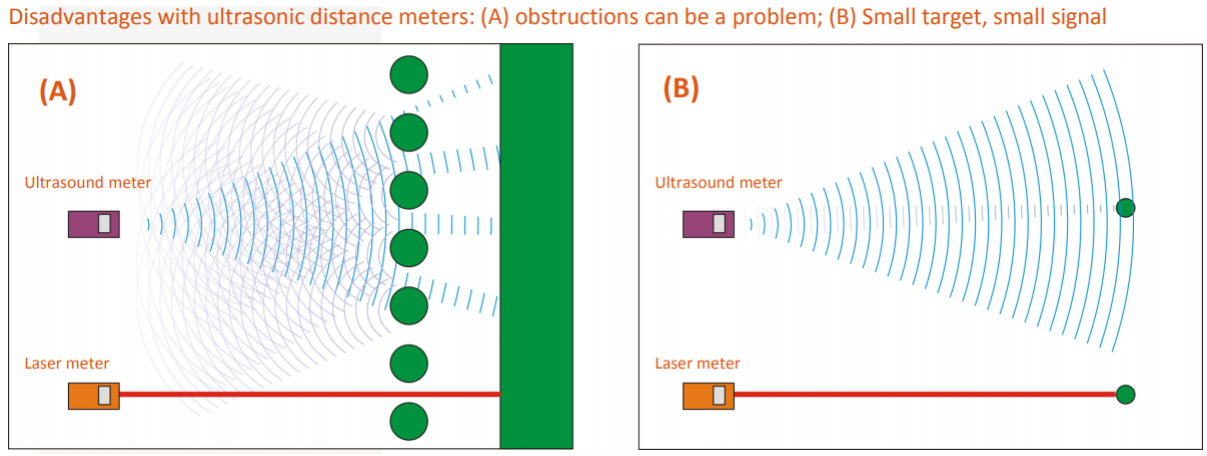

FIGURE 3.5: Ultrasonic sensor

3.4 Force sensor

There are many types of force sensors. A 1N Force will cause 1 kg mass to accelerate at $1m/s^2$. Force sensing (solids) and Pressure sensing (liquids and gases) are tied together. Pressure sensing requires the measurement of force (P=F/Area). The pressure is Force distributed over a large area whereas Force is concentrated on a spot. Types of force sensors:

- Strain Gauge
- Tactile Sensors
- Touch Sensors
- Piezoelectric Sensors

A strain gauge can be used to measure force accurately. There are many different types of strain gauges. A strain gauge can be made from capacitors and inductors, but the most widely used types are made from resistors. A wire strain gauge is made from a resistor, in the form of a metal foil. The principle of operation is that the resistance of a wire increases with increasing strain and decreases with decreasing strain. In order to measure strain with a strain gauge, it must be connected to an electrical circuit, and a Wheatstone bridge is commonly used to detect the small changes in the resistance of the strain gauge. Strain gauges can be used to measure force, load, weight pressure, torque or displacement.

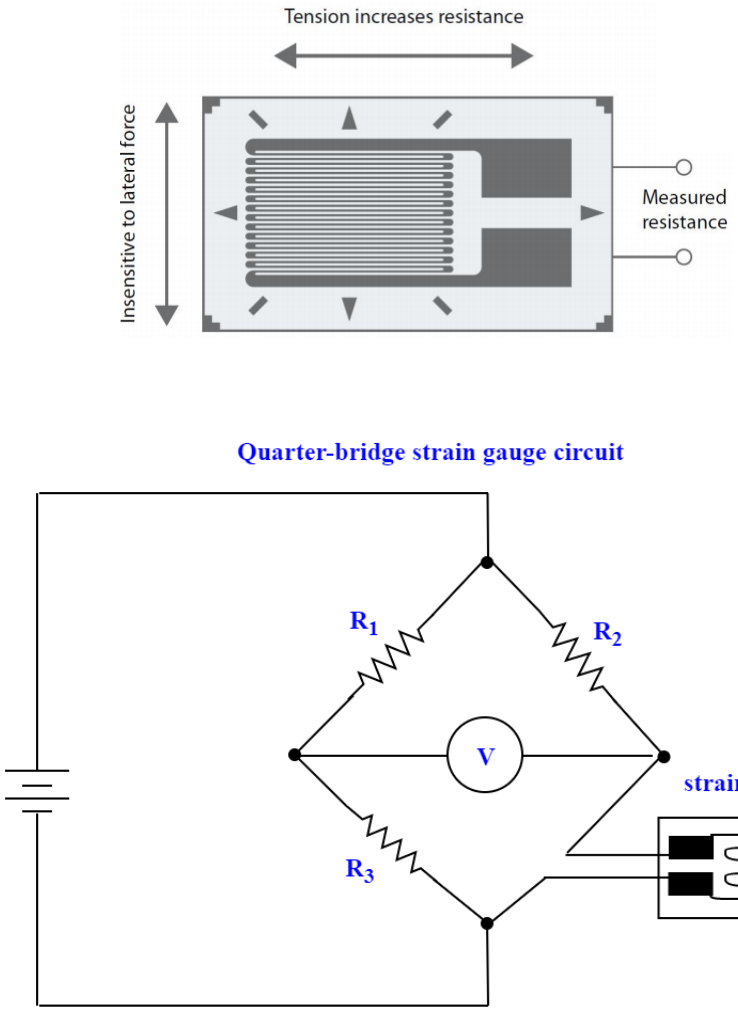

FIGURE 3.6: Strain gauge sensor

Force can also be measured using the principle of piezoelectricity. A piezoelectric sensor produces a voltage when a force is applied to its surface. The disadvantage of this method is that the voltage decays after the application of the force and thus piezoelectric sensors are only useful for measuring dynamic force.

3.5 Velocity and acceleration sensors

Velocity is the differentiation of position, and in general position sensors can be used to measure velocity. The required differentiation can be done either in hardware (e.g. using operational amplifiers) or by the computer.
There are two types of velocity sensors: linear sensors, and rotary sensors. Linear velocity sensors can be constructed using a pair of coils and a moving magnet. When the coils are connected in series, the movement of the magnet produces an additive voltage which is proportional to the movement of the magnet. One of the most widely used rotary velocity

sensors is the tachometer (or tachogenerator). A tachometer is connected to the shaft of a rotating device (e.g. a motor) and produces an analog voltage that is proportional to the speed of the shaft. If ω is the angular velocity of the shaft, the output voltage of the tachometer is given by

$$V_o = k\omega \tag{3.2}$$

where k is the gain constant of the tachometer.

Another popular velocity sensor is the optical encoder. This basically consists of a light source and a disk with opaque and transparent sections where the disk is attached to the rotating shaft. A light sensor at the other side of the wheel detects light and a pulse is produced when the transparent section of the disk comes round. The encoder's controller counts the pulses in a given time, and this is proportional to the speed of the shaft.

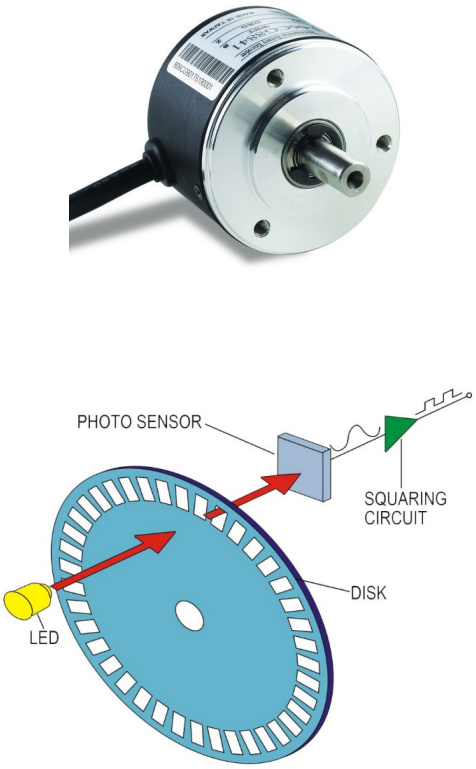

FIGURE 3.7: Rotary encoder

Acceleration is the differentiation of velocity or the double differentiation of position. Thus, in general, position sensors can be used to measure acceleration. The differentiation can be done either by using operational amplifiers or by a computer program. For accurate measurement of the acceleration, semiconductor accelerometers can be used.

Review questions

1. What is the difference between active and non-active sensors?
2. What is the difference between digital and analog sensors?
3. Make a small experiment. But thermistor on the hot water and take a characteristic of thermister.
4. Make a small experiment. Using potentiometer draw a characteristic between angle and resistor value.
5. Calculate the distance of the ultrasonic sensor if the Time of Flight was 3700 us.
6. Explain how to work a strain gauge sensor. Draw quarter bridge strain gauge circuit.
7. Explain how to work a rotary encoder sensor. Is it possible to measure the acceleration of the object?
8. What kind of sensor can be used to control a traffic light?
9. Draw a block diagram of a closed-loop system with a force sensor? What king system could it be?
10. Calculate the distance of the laser sensor if the Time of Flight was 900 us.

CHAPTER 4

Electric motors

Topic 4

Electric motors

- DC motor vs AC motor and Stepper motor
- Bridge control
- Pulse Width Modulation
- Optoisolator

An **electric motor** is an electrical machine that converts electrical energy into mechanical energy [5]. Most electric motors operate through the interaction between the motor's magnetic field and electric current in a wire winding to generate force in the form of rotation of a shaft. Electric motors can be powered by direct current (DC) sources, such as from batteries, motor vehicles or rectifiers, or by **alternating current** (AC) sources, such as a power grid, inverters or electrical generators. An **electric generator** is mechanically identical to an electric motor, but operates in the reverse direction, converting mechanical energy into electrical energy.

4.1 DC motor control

A **direct current** (DC) motor is a widely used device that translates electrical pulses into mechanical movement. In the DC motor, we have only + and - leads. Connecting them to a DC voltage source moves the motor in one direction. By reversing the polarity, the DC motor will move in the opposite direction. For example, the small fans used in many motherboards to cool the CPU are run by DC motors. When the leads are connected to the + and - voltage source, the DC motor moves continuously.
The maximum speed of a DC motor is indicated in rpm (**revolutions per minute**) and is given in the datasheet. The DC motor has two rpms: no-load and loaded. The manufacturer's datasheet gives the no-load rpm. The no-load rpm can be from a few thousand to tens of thousands. The rpm is reduced when moving a load and it decreases as the load is increased.
For example, a drill turning a screw has a much lower rpm speed than when it is in the no-load situation. DC motors also have voltage and current ratings. The nominal voltage is the voltage for that motor under normal conditions and can be from 1 to 150 V, depending on the motor. As we increase the voltage, the rpm goes up. The current rating is the current consumption when the nominal voltage is applied with no load and

can be from 25 mA to a few amps. As the load increases, the rpm is decreased, unless the current or voltage provided to the motor is increased, which in turn increases the torque. With a fixed voltage, as the load increases, the current (power) consumption of a DC motor is increased. If we overload the motor it will stall, and that can damage the motor due to the heat generated by high current consumption. Figure 4.1 shows the DC motor rotation for clockwise (CW) and counter-clockwise (CCW) rotations.

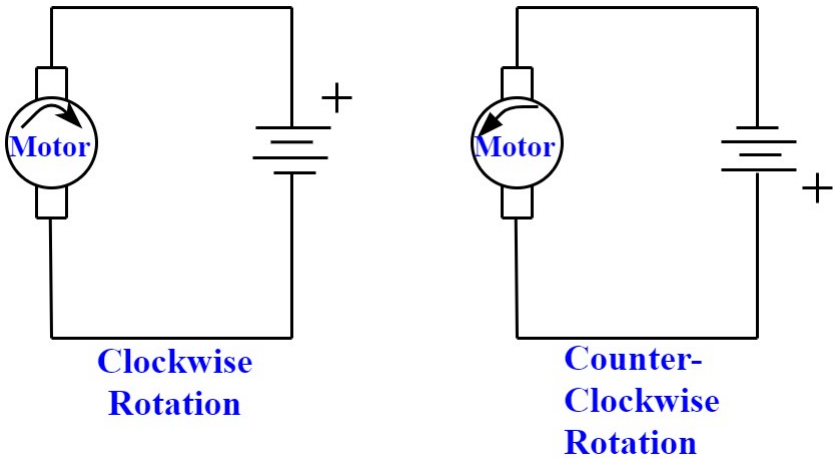

FIGURE 4.1: DC motor rotation

With the help of relays or some specially designed chips we can change the direction of the DC motor rotation. Figure 4.2 show the basic concepts of H-bridge control of DC motors.

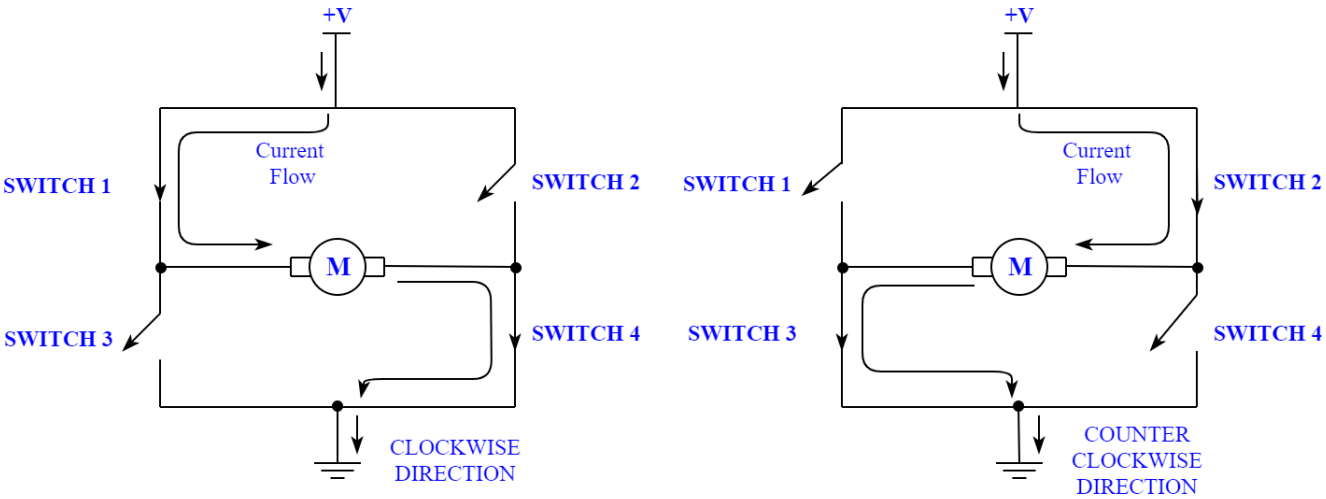

FIGURE 4.2: H-bridge control

Figure 4.3 shows an invalid configuration. Current flow directly to ground, creating a **short circuit**. The same effect occurs when switches 1 and 3 are closed or switches 2 and 4 are closed.

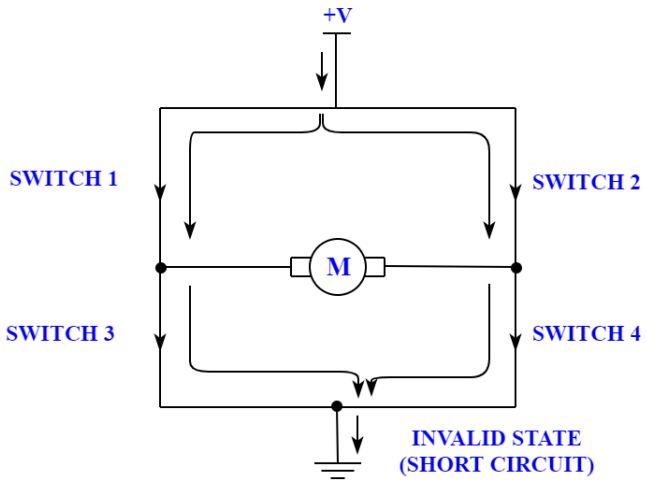

FIGURE 4.3: H-bridge invalid configuration

Motor operation	SW1	SW2	SW3	SW4
Off	Open	Open	Open	Open
Clockwise	Closed	Open	Open	Closed
Counterclockwise	Open	Closed	Closed	Open
Invalid	Closed	Closed	Closed	Closed

Need to remember

The **optoisolator** is indispensable in many motor control applications. Notice that the AVR is protected from EMI created by motor brushes by using an optoisolator and a separate power supply. Figure 4.4 show optoisolators for single directional motor control, and the same principle should be used for most motor applications. Separating the power supplies of the motor and logic will reduce the possibility of damage to the control circuit. Figure 4.4 shows the connection of a bipolar transistor to a motor. The protection of the control circuit is provided by the optoisolator. The motor and AVR use separate power supplies.

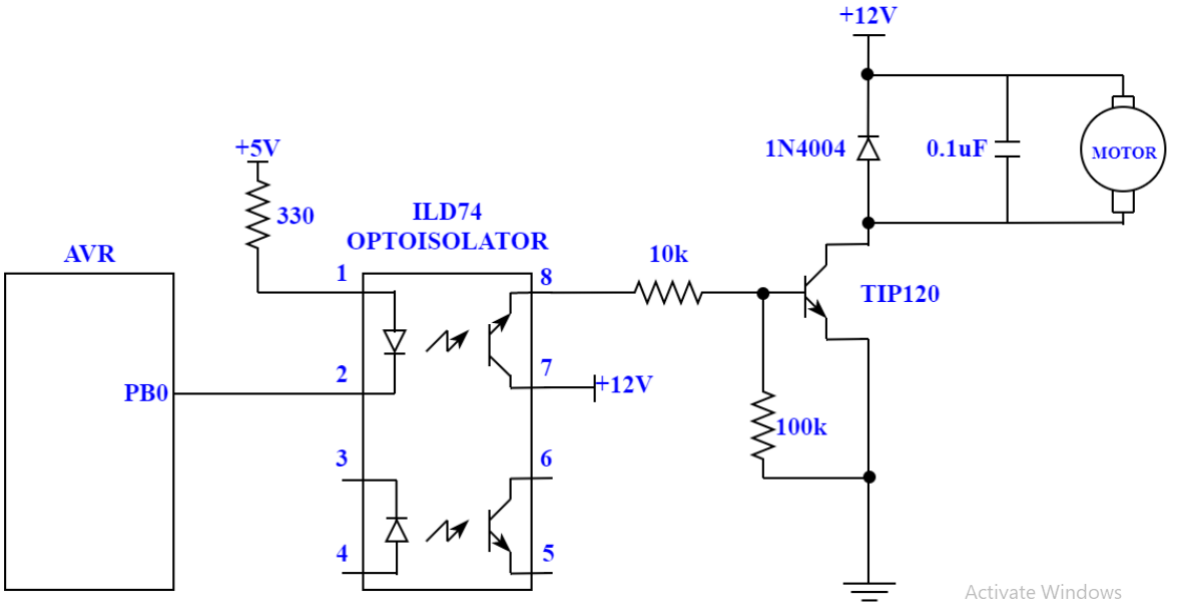

FIGURE 4.4: DC motor with optoisolator

The separation of power supplies also allows the use of high-voltage motors. Notice that we use a decoupling capacitor across the motor, this helps reduce the EMI created by the motor. The motor is switched on by clearing bit PB0.

The speed of the motor depends on three factors: (a) load, (b) voltage, and (c) current. For a given fixed load we can maintain a steady speed by using a method called **pulse width modulation** (PWM). By changing (modulating) the width of the pulse applied to the DC motor we can increase or decrease the amount of power provided to the motor, thereby increasing or decreasing the motor speed.

Notice that, although the voltage has a fixed amplitude, it has a variable duty cycle. That means the wider the pulse, the higher the speed. PWM is so widely used in DC motor control that some microcontrollers come with the PWM circuitry embedded in the chip. In such microcontrollers, all we have to do is load the proper registers with the values of the high and low portions of the desired pulse, and the rest is taken care of by the microcontroller. This allows the microcontroller to do other things. For microcontrollers without PWM circuitry, we must create the various duty cycle pulses using software, which prevents the microcontroller from doing other things.

The ability to control the speed of the DC motor using PWM is one reason that DC motors are often preferred over AC motors.

4.2 Stepper motor control

Asynchronous and brushless DC motor that converts electrical pulses into mechanical movements and thus, rotates stepwise with a certain angle between each step for completing a full rotation is called as Stepper Motor. The angle between the steps of rotation of the stepper motor is termed as the stepper angle of the motor.
Stepper motors are DC brushless motors that can rotate from 0° to 360° in steps. Stepper motor uses electronic signals to rotate the motor in steps and each signal rotates the shaft in the fixed increment (one step). The rotation angle is controlled by applying a certain sequence of signals. Unlike Servo motor, stepper motors can be driven by using GPIO pins of microcontroller rather than PWM pins and can rotate in (+360°) and (-360°).
The order of signals decides the clockwise and counter-clockwise direction of the stepper motor. To control the speed of the motor, we just need to change the rate of control signals applied.
The stepper motors rotate in steps. This is very useful because it can be precisely positioned without any feedback sensor, which represents an open-loop controller. The stepper motor consists of a rotor that is generally a permanent magnet and it is surrounded by the windings of the stator.
As we activate the windings step by step in a particular order and let a current flow through them they will magnetize the stator and make electromagnetic poles respectively that will cause propulsion to the motor. So that is the basic working principle of the stepper motors.
There are several modes of steps to operate Stepper Motor such as full step, half step and microstep.
The first one is the **Wave Drive** or Single-Coil Excitation. In this mode we active just one coil at a time which means that for this example of a motor with 4 coils, the rotor will make full cycle in 4 steps.

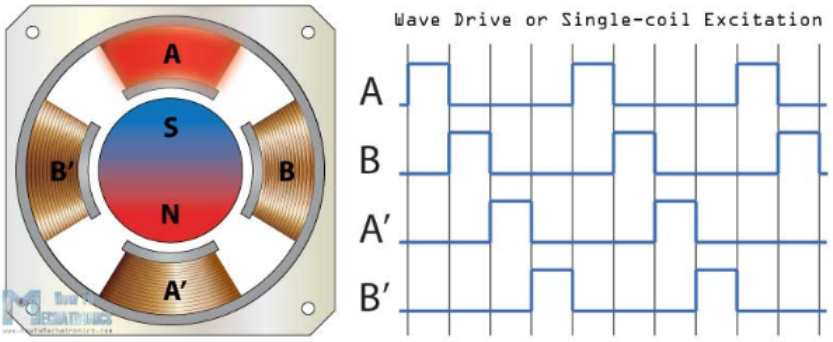

FIGURE 4.5: Wave drive

The **Full Step Drive** mode which provides much higher torque output because we always have 2 active coils at a given time. However, this doesn't improve the resolution of the stepper and again the rotor will make a full cycle in 4 steps.

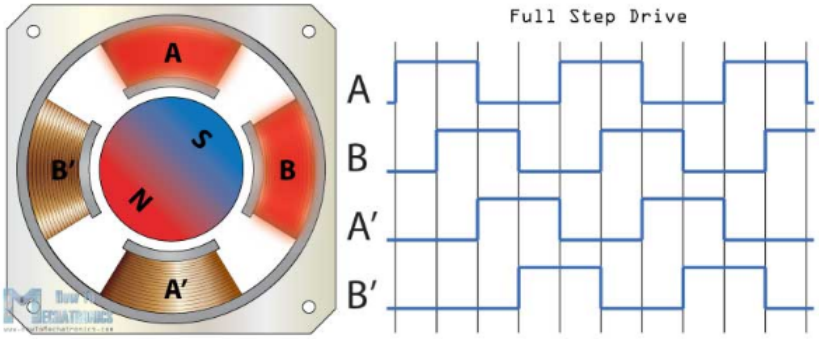

FIGURE 4.6: Full step

For increasing the resolution of the stepper we use the **Half Step Drive** mode. This mode is actually a combination of the previous two modes. Here we have one active coil followed by 2 active coils and then again one active coil followed by 2 active coils and so on. So with this mode, we get double the resolution with the same construction. Now the rotor will make full cycle in 8 steps.

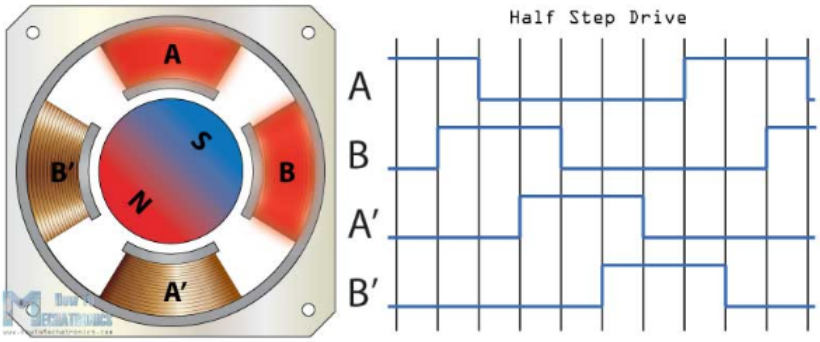

FIGURE 4.7: Half step

However, the most common method of controlling stepper motors nowadays is the **Microstepping**. In this mode, we provide variable controlled current to the coils in the form of a sin wave. This will provide smooth motion of the rotor, decrease the stress of the parts and increase the accuracy of the stepper motor.

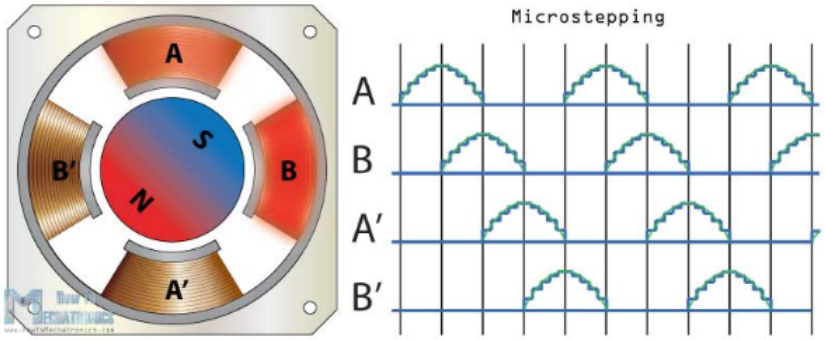

FIGURE 4.8: Micro step

Another way of increasing the resolution of the stepper motor is by increasing the numbers of the poles of the rotor and the numbers of the pole of the stator.

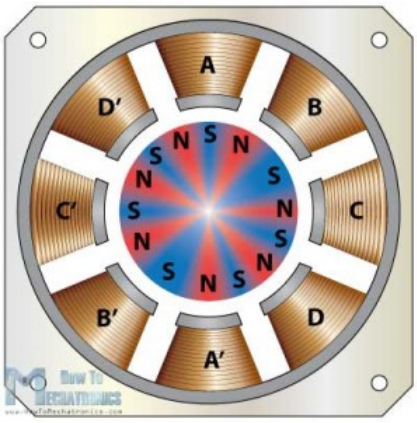

FIGURE 4.9: Number of poles

Important

Stepper motor consumes high current and hence, we will need to use a driver IC like the **ULN2003** in order to control the motor with a microcontroller like the AVR. Known for its high current and high voltage capacity, the ULN2003 gives a higher current gain than a single transistor and enables the low voltage and low current output of a microcontroller to drive a higher current stepper motor. For example, a stepper motor that needs 9V and 300mA to operate cannot be powered by an AVR. Hence, we connect this IC to the source for enough current and voltage for the motor. If you have to power anything more than 5V and 40mA, the ULN2003 driver board should be used.

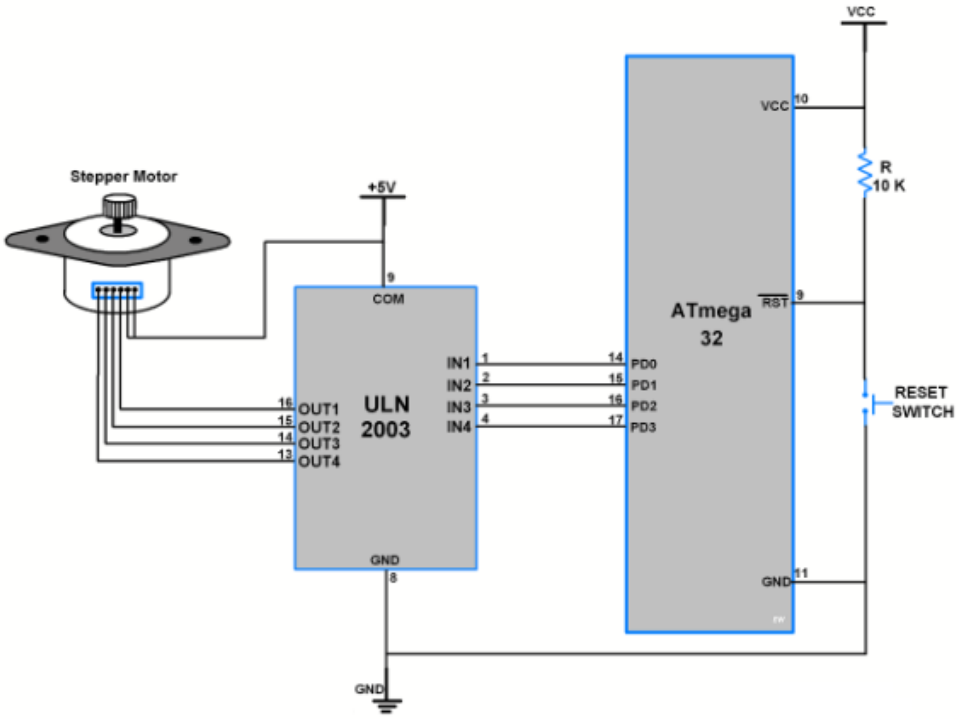

FIGURE 4.10: Stepper motor with ULN2003 connection

The ULN2003 is one of the most common motor driver ICs that houses an array of 7 Darlington transistor pairs, each capable of driving loads up to 500mA and 50V. Basically, a Darlington pair is a pair of transistors, where the second transistor amplifies the output current of the first transistor. The ULN2003 IC is needed to drive the motor with an AVR.

Review questions

1. What happens if we change the polarity of the DC motor?
2. Draw H bridge control for DC motor. Explain how it works.
3. What is the optoisolator and how to use it to control the DC motor?
4. Draw the circuit with the AVR controller to control the DC motor direction.
5. What is the difference between the DC and Stepper motors?
6. Draw time diagram of Full step drive.
7. Why we use ULN2003 for controlling the Stepper motor?
8. What is the mean difference between AC and DC motors?
9. How we can change the rotation speed of the DC motor?
10. How can we control the stability of rotation speed?

CHAPTER 5

Microprocessors and Microcontrollers

Topic 5

Processor systems

- Processor architecture
- Microprocessors system
- Microprocessors vs Microcontroller
- Buses

A **microprocessor** is an electronic component that is used by a computer to do its work. It is a Central Processing Unit on a single integrated circuit chip containing millions of very small components including transistors, resistors, and diodes that work together [6]. The microprocessor is a multipurpose, clock driven, register based, digital integrated circuit that accepts binary data as input, processes it according to instructions stored in its memory and provides results (also in binary form) as output. Microprocessors contain both combinational logic and sequential digital logic. Microprocessors operate on numbers and symbols represented in the binary number system.

5.1 Processor Architecture

A **processor** is the brain of a computer which basically consists of Arithmetical and Logical Unit (ALU), Control Unit and Register Array.

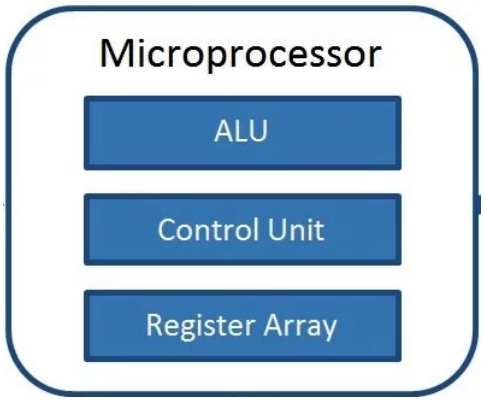

FIGURE 5.1: Microprocessor

5.2 General Purpose Register

In computer architecture, a **processor register** is a quickly accessible location available to a computer's central processing unit (CPU). Registers usually consist of a small amount of fast storage, although some registers have specific hardware functions, and maybe read-only or write-only. Registers are typically addressed by mechanisms other than the main memory. Processor registers are normally at the top of the memory hierarchy and provide the fastest way to access data.

Registers are normally measured by the number of bits they can hold, for example, an "8-bit register", "32-bit register" or a "64-bit register" or even more. A processor often contains several kinds of registers, which can be classified according to their content or instructions that operate on them:

• User-accessible registers can be read or written by machine instructions. The most common division of user-accessible registers is into data registers and address registers.

• Data registers can hold numeric data values such as integer and, in some architectures, floating-point values, as well as characters, small bit arrays, and other data. In some older and low-end CPUs, a special data register, known as the accumulator, is used implicitly for many operations.

• Address registers hold addresses and are used by instructions that indirectly access primary memory. Some processors contain registers that may only be used to hold an address or only to hold numeric values (in some cases used as an index register whose value is added as an offset from some address), others allow registers to hold either kind of quantity. A wide variety of possible addressing modes, used to specify the effective address of an operand, exist.

The stack pointer is used to manage the run-time stack. Rarely, other data stacks are addressed by dedicated address registers, see stack machine.

• General-purpose registers (GPRs) can store both data and addresses, i.e., they are combined data/address registers and rarely the register file is unified to include floating point as well.

• Status registers hold truth values often used to determine whether some instruction should or should not be executed.

• Floating-point registers (FPRs) store floating-point numbers in many architectures.

• Constant registers hold read-only values such as zero, one, or pi.

• Vector registers hold data for vector processing done by SIMD instructions (Single Instruction, Multiple Data).

• Special-purpose registers (SPRs) hold program state, they usually include the program counter, also called the instruction pointer, and the status register, the program counter

and status register might be combined in a program status word (PSW) register. The aforementioned stack pointer is sometimes also included in this group. Embedded microprocessors can also have registers corresponding to specialized hardware elements.

• In some architectures, model-specific registers (also called machine-specific registers) store data and settings related to the processor itself. Because their meanings are attached to the design of a specific processor, they cannot be expected to remain standard between processor generations.

• Internal registers – registers not accessible by instructions, used internally for processor operations.

• Instruction register, holding the instruction currently being executed.

5.3 Arithmetic Logic Unit

An **arithmetic logic unit** (ALU) is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer. Examples of arithmetic operations are addition, subtraction, multiplication, and division. Examples of logic operations are comparisons of values such as NOT, AND, and OR.

5.4 Control Unit

The **control unit** (CU) is a component of a computer's central processing unit (CPU) that directs the operation of the processor. It tells the computer's memory, arithmetic and logic unit and input and output devices how to respond to the instructions that have been sent to the processor. It directs the operation of the other units by providing timing and control signals. Most computer resources are managed by the CU. It directs the flow of data between the CPU and the other devices.

5.5 Microprocessors system

An alone microprocessor cannot do anything. So it needs to corporate with other devices to perform some job. The purpose of a microprocessor is to perform mathematical calculations (computations) in an artificial manner. The combination of a microprocessor with memory and peripherals results in a system that is designed to be useful for controlling other devices or for performing computations quickly. All microprocessor systems perform the same essential functions, that is, data or signal input, storage, processing, and output.

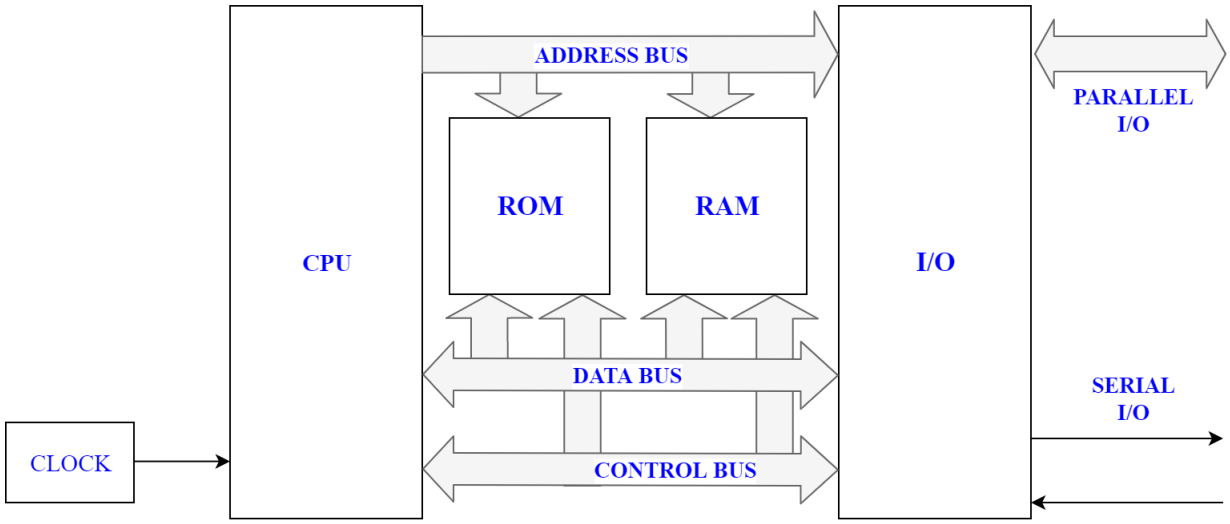

FIGURE 5.2: Processor system

5.6 Buses

A **bus** is a group of conducting wires which carries information, all the peripherals are connected to microprocessor through Bus. There are three types of buses.

Need to remember

- **Address bus**

It is a group of conducting wires which carry address only. The address bus is unidirectional because data flow in one direction, from the microprocessor to memory or from the microprocessor to Input/Output devices. The Length of the address bus determines the amount of memory a system can address. Such as a system with a 32-bit address bus can address 2^{32} memory locations. If each memory location holds one byte, the addressable memory space is 4 GB.

• **Data bus**

It is a group of conducting wires which carry Data only. The data bus is bidirectional because of data flow in both directions, from the microprocessor to memory or Input/Output devices and from memory or Input/Output devices to microprocessors. The width of the data bus is directly related to the largest number that the bus can carry, such as an 8-bit bus can represent 2 to the power of 8 unique values, this equates to the number 0 to 255. A 16-bit bus can carry 0 to 65535.

• **Control bus**

It is a group of conducting wires, which is used to generate timing and control signals to control all the associated peripherals, the microprocessor uses control bus to process data that is what to do with the selected memory location. Some control signals are:

- Memory read
- Memory write
- I/O read
- I/O write
- Opcode fetch

If one line of control bus may be the read/write line. If the wire is low (no electricity flowing) then the memory is read, if the wire is high (electricity is flowing) then the memory is written.

5.7 Microcontroller internal structure

A **microcontroller** is a computer system on a chip that does a job. It contains an integrated processor, memory, and programmable Input/Output peripherals, which are used to interact with things connected to the chip. A microcontroller is different than a microprocessor, which only contains a CPU.

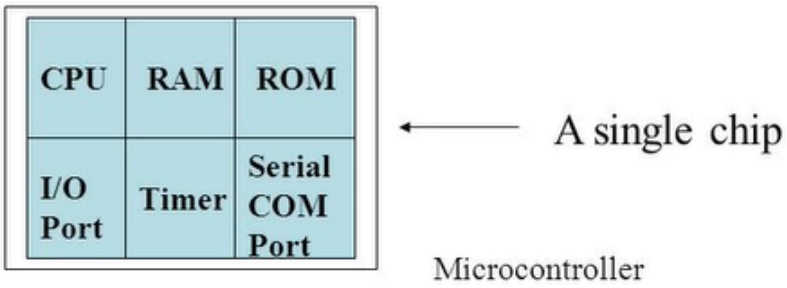

FIGURE 5.3: Microcontroller

The core elements of a microcontroller are:

• **The processor** (CPU) - A processor can be thought of as the brain of the device. It processes and responds to various instructions that direct the microcontroller's function. This involves performing basic arithmetic, logic and I/O operations. It also performs data transfer operations, which communicate commands to other components in the larger embedded system.

• **Memory** - A microcontroller's memory is used to store the data that the processor receives and uses to respond to instructions that it's been programmed to carry out. A microcontroller has two main memory types:

1. **Program memory**, which stores long-term information about the instructions that the CPU carries out. Program memory is non-volatile memory, meaning it holds information over time without needing a power source.

2. **Data memory**, which is required for temporary data storage while the instructions are being executed. Data memory is volatile, meaning the data it holds is temporary and is only maintained if the device is connected to a power source.

• **I/O peripherals** - The input and output devices are the interface for the processor to the outside world. The input ports receive information and send it to the processor in the form of binary data. The processor receives that data and sends the necessary instructions to output devices that execute tasks external to the microcontroller.

Microprocessor	**Microcontroller**
It is a central processing unit on a single silicon-based integrated chip.	It is a byproduct of the development of microprocessor with a CPU along with other peripherals.
It has no RAM, ROM, IO units, Timers and other peripherals on the chip.	It has a CPU along with RAM, ROM, and other peripherals embedded on a single chip.
It is the heart of the computer system.	It is the brains of the computer system.
It uses an external bus to interface to RAM, ROM, IO pins, and other peripherals	It used an internal controlling bus which is not available to the board designer
Microprocessor based systems can run at very high speed because of the technology involved.	Microcontroller based systems run up to 200MHz or more depending on the architecture.
It's used for general purpose applications which are able to handle loads of data	It's used for application specific systems.
It's complex and expensive with large number of instructions to process.	It's simple and inexpensive with less number of instructions to process.

Review questions

1. What are the main components in the microprocessor?
2. What is the main difference between microprocessor and processor systems?
3. What kind of buses in the processor system uses?
4. Draw a block diagram of the processor system.
5. What is the main difference between microprocessor and microcontroller?
6. If the address bus can access 4GB memory, how many bits on this bus?
7. If the data bus has 16 bit how many kinds of data can it send?
8. Explain how to work ALU?
9. What is the main difference between register and memory?
10. What is the main difference between data memory and program memory?

CHAPTER 6

Programming in C

Topic 6

Programming in C

- Assembly language vs C
- Variable types and sizes
- Program flow control
- Functions

Microcontrollers have traditionally been programmed using the **assembly language** [5, 7]. This language consists of various mnemonics which describe the instructions of the target microcontroller. An assembly language is unique to a microcontroller and cannot be used for any other type of microcontroller. Although the assembly language is very fast, it has some major disadvantages. Perhaps the most important of these is that the assembly language can become very complex and difficult to maintain. It is usually a very time-consuming task to develop large projects using the assembly language. Program debugging and testing are also considerably more complex, requiring more effort and more time.
Microcontrollers can also be programmed using high-level languages. C programming is less time consuming and much easier to write, but the hex file size produced is much larger than if we used Assembly language.

Need to remember

The following are some of the major reasons for writing programs in C instead of Assembly:

- It is easier and less time consuming to write in C than in Assembly.
- C is easier to modify and update.
- You can use code available in function libraries,
- C code is portable to other microcontrollers with little or no modification.

6.1 Variables

One of the goals of AVR programmers is to create smaller hex files, so it is worthwhile to re-examine C data types. In other words, a good understanding of C data types for the AVR can help programmers to create smaller hex files. In this section we focus on the specific C data types that are most common and widely used in MR C compilers, Table 7-1 shows data types and sizes, but these may vary from one compiler to another. The compiler supports the following basic data types:

- bit
- unsigned char
- signed char
- unsigned int
- signed int
- long
- unsigned long
- float
- double

Data Type	Size in Bits	Data Range
unsigned char	8-bit	0 to 255
char	8-bit	-128 to +128
unsigned int	16-bit	0 to 65535
int	6-bit	-32768 to +32767
unsigned long	32-bit	0 to 4294967295
long	32-bit	-2147483648 to +2147483648
float	32-bit	± 1.175 e-38 to ± 3.402e38
double	32-bit	± 1.175 e-38 to ± 3.402e38

Function signature	Description of
int atoi(char *str)	Converts the string str to integer.
long atol	Converts the string str to long.
void itoa(int n, char *str)	Converts the integer n to characters in string str.
void ltoa (int n, char *str)	Converts the long n to characters in string str.
float atof (char *str)	Converts the characters from string str to float

6.2 Program flow Control

The AVR language supports the following flow control commands:

- if-else
- for
- while
- do
- goto
- break
- continue
- switch–case

6.2.1 If –Else statement

This statement is used to decide whether or not to execute a single statement or a group of statements depending upon the result of a test. There are several formats for this statement, the simplest one is when there is only one statement:

```
1 if (condition) statement;
```

The following test decides whether a student has passed an exam with a pass mark of 45 and if so, character 'P' is assigned to the variable student:

```
1 if (result > 45) student = 'P';
```

In the multiple-statement version of the *if* statement, the statements are enclosed in curly brackets as in the following format:

```
1 if (condition)
2 {
3 statement;
4 statement;
5 ...
6 statement;
7 statement;
8 }
```

For example,

```
1 if (temperature > 20)
2 { flag = 1; pressure = 20; hot = 1; }
```

The *if* statement can be used together with the *else* statement when it is required to execute an alternative set of statements when a condition is not satisfied. The general format is:

```
if(condition)
{
statement;
statement;
...
statement;
statement;
}
else
{
statement;
statement;
...
statement;
statement;
}
```

In the following example, if the result is greater than 50 the variable student is assigned character 'P' and count is incremented by 1. Otherwise (i.e. if the result is less than or equal to 50) the variable student is assigned character 'F' and count is decremented by 1:

```
if(result > 50)
{
student = 'P';
count++;
}
else
{
student = 'F';
 count--;
}
```

When using the equals sign as a condition, double equals signs '==' should be used as in the following example:

```
if(total == 100)
x++;
else
y++;
```

6.2.2 Switch Case

This is another form of flow control where statements are executed depending on a multi-way decision. The *switch − case* statement can only be used in certain cases where:

- only one variable is tested and all branches depend on the value of that variable;
- each possible value of the variable can control a single branch.

The general format of the *switch − case* statement is as follows. Here, a variable *number* is tested. If a *number* is equal to n1, statements between n1 and n2 are executed. If *number* is equal to n2, statements between n2 and n3 are executed, and soon. If *number* is not equal to any of the condition then the statements after the default case is executed. Notice that each block of statement is terminated with a *break* statement so that the program jumps out of the *switch − case* block.

```
switch (number)
{
case n1:
statement;
...
statement;
break;
case n2:
statement;
...
statement;
break;
case n3:
statement;
...
statement;
break;
default:
statement;
...
statement;
break;
}
```

In the following example, the variable *no* stores a hexadecimal number between A and F and the *switch* − *case* statement is used to convert the hexadecimal number to a decimal number in the variable:

```
switch (no)
{
case 'A':
 deci = 65;
 break;
 case 'B':
 deci = 66;
 break;
 case 'C':
 deci = 67;
 break;
 case 'D':
 deci = 68;
 break;
 case 'E':
deci = 69;
break;
case 'F':
deci = 70;
break;
}
```

6.2.3 For statement

The *for* statement is used to create loops in programs. The *for* loop works well where the number of iterations of the loop is known before the loop is entered. The general format of the for statement is:

```
for (initial; condition; increment)
{
 statement;
 statement;
}
```

The first parameter is the initial condition and the loop is executed with this initial condition being true. The second is a test and the loop is terminated when this test returns a false. The third is a statement that is executed every time the loop body is completed. This is usually an increment of the loop counter. An example is given below where the statements inside the loop are executed 10 times. The initial value of the

variable i is zero and this variable is incremented by one every time the body of the loop is executed. The loop is terminated when i become 10 (i.e. the loop is executed 10 times):

```
for (i = 0; i < 10; i++)
{
 sum++;
 total = total + sum;
}
```

The above code can also be written as follows, where the initial value of i is 1:

```
for (i = 1; i <= 10; i++)
{
 sum++;
total = total + sum;
}
```

If there is only one statement to be executed, the *for* loop can be written as in the following example:

```
for(i = 0; i < 10; i++)
count++;
```

It is possible to declare nested *for* loops where one loop can be inside another loop. An example is given below where the inner loop is executed 5 times, and the outer loop is executed 10 times:

```
for(i = 0; i < 10; i++)
{
    cnt++;
    for(j = 0; j < 5; j++)
    {
        sum++;
    }
}
```

6.2.4 While statement

The *while* loop repeats a statement until the condition at the beginning of the statement becomes false. The general format of this statement is:

```
while (condition) statement;
```

or

```
while (condition)
{
statement;
statement;
 ...
statement;
}
```

In the following example, the loop is executed 10 times:

```
i = 0;
while (i < 10)
{
 cnt;
 total = total + cnt;
 i++;
}
```

Notice that the condition at the beginning of the loop should become true for the loop to terminate, otherwise we get an infinite loop as shown in the following example:

```
i = 0;
while (i < 10)
{
cnt;
 total = total + cnt;
}
```

Here the variable i is always less than 10 and the loop never terminates.

6.2.5 Do statement

This is another form of the *while* statement where the condition to terminate the loop is tested at the end of the loop and, as a result, the loop is executed at least once. The condition to terminate the loop should be satisfied inside the loop, otherwise, we get an infinite loop. The general format of this statement is:

```
do
{
statement;
statement;
 ...
 statement;
 }
 while (condition);
```

An example is given below where the loop is executed five times:

```
j = 0;
do
 {
 cnt++;
 j++;
 }
while (j < 5);
```

6.2.6 Break statement

We have seen the use of this statement in the $switch-case$ blocks to terminate the block. Another use of the *break* statement is to terminate a loop before a condition is met. An example is given below where the loop is terminated when the variable j becomes 10:

```
while(i < 100)
{
 total++;
 sum = sum + total;
 if (j = 10) break;
}
```

6.2.7 Continue statement

The *continue* statement is similar to the *break* statement but is used less frequently. The *continue* statement causes a jump to the loop control statement. In a *while* loop, control jumps to the condition statement, and in a for loop, control jumps to the beginning of the loop.

6.3 Functions

Almost all programming languages support **functions** or some similar concepts. Some languages call them **subroutines**, some call them **procedures**. Some languages distinguish between functions that return variables and those which do not. In almost all programming languages functions are of two kinds: **user functions** and **built-in functions**. User functions are developed by programmers, while built-in functions are usually general purpose routines provided with the compiler. Functions are independent program codes and are usually used to return values to the main calling programs.
These functions are developed by the programmer. Every function has a name and optional arguments, and a pair of brackets must be used after the function name in order to declare the arguments. The function performs the required operation and can return values to the main calling program if required. Not all functions return values. Functions whose names start with the keyword *void* do not return any values, as shown in the following example:

```
void led on()
{
led = 1;
}
```

The return value of a function is included inside a return statement as shown below. This function is named sum, has two integer arguments named a and b, and the function returns an integer:

```
int sum (int a, int b)
{
int z;
z = a + b;
return (z);
}
```

A function is called in the main program by specifying the name of the function and assigning it to a variable. In the following example, the variable w in the main program is assigned the value 7:

```
w = sum (3, 4)
```

It is important to realize that the variables used inside a function are local to that function and do not have any relationship to any variables used outside the function with the same names.

Review questions

1. What is the difference between Assembler and C languages?
2. Fill the bits.

a. unsigned char - ...

b. int - ...

c. long - ...

d. float - ...

e. double - ...

3. What is the function that converts an integer to char?
4. Write code whether a student has passed an exam using if statement.
5. The toaster is a closed-loop system that uses a thermistor on the feedback loop. Draw a program algorithm of that system.
6. Write code that selects a number using switch case.
7. Write code that blinks light diode 100 times using for statement.
8. Write code that blinks light diode 100 times using *while* statement.
9. Write program function of function $y = 5 * x$.
10. What is the difference between *while* and *do while*?

CHAPTER 7

Introduction to AVR controllers

Topic 7

AVR controller

- Atmega32 controller
- Memory organization
- AVR internal devices
- Programming registers

AVR features The AVR is an 8-bit RISC single-chip microcontroller with Harvard architecture that comes with some standard features such as on-chip program ROM, data RAM, data EEPROM, timers and I/O ports [5, 8, 9, 10, 11, 12, 13]. Most AVRs have some additional features like ADC, PWM, and different kinds of serial interfaces such as USART, SPI, I2C (TWI), CAN, USB, and so on.

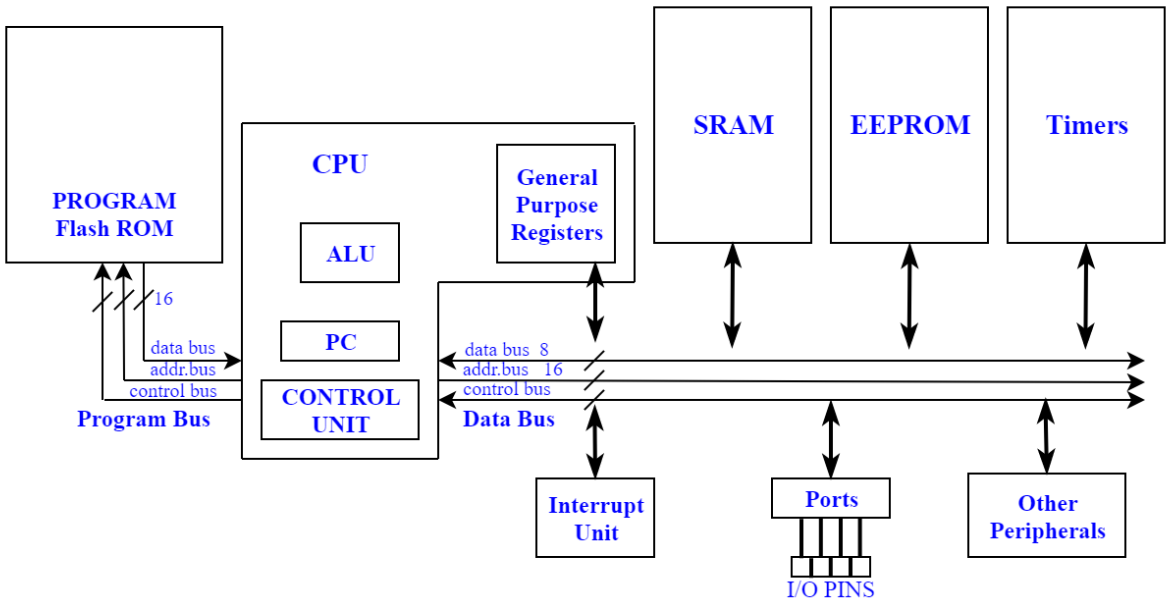

FIGURE 7.1: AVR microcontroller

7.1 Memory organization

The ROM is used to store program code, the RAM space is for data storage. The AVR has a maximum of 64K bytes of data RAM space. Not all of the family members come with that much RAM.

Need to remember

The data RAM space has three components: general-purpose registers, I/O memory, and internal SRAM. There are 32 general-purpose registers in all of the AVRs, but the SRAM's size and the I/O memory's size varies from chip to chip. In AVR, we also have a small amount of EEPROM to store critical data that does not need to be changed very often.

Part Num.	Code ROM	Data RAM	Data EEPROM	IO pinc	ADC	Timers
ATmega8	8K	1K	0.5K	23	8	3
ATmega16	16K	1K	0.5K	32	8	3
ATmega32	32K	2K	1K	32	8	3
ATmega64	64K	4K	2K	54	8	4
ATmega1280	128K	8K	4K	86	16	6

In AVR there are 32 general purpose registers. They are R0-R31 and are located in the lowest location of the memory address. All of these registers are 8 bits. The general purpose registers in AVR are the same as the accumulator in other microprocessors. They can be used by all arithmetic and logic instructions. In AVR microcontrollers there are two kinds of memory space: **code memory space** and **data memory space**. Our program is stored in code memory space, whereas the data memory stores data. The data memory is composed of three parts: GPRs (general purpose registers), I/O memory, and internal data SRAM.

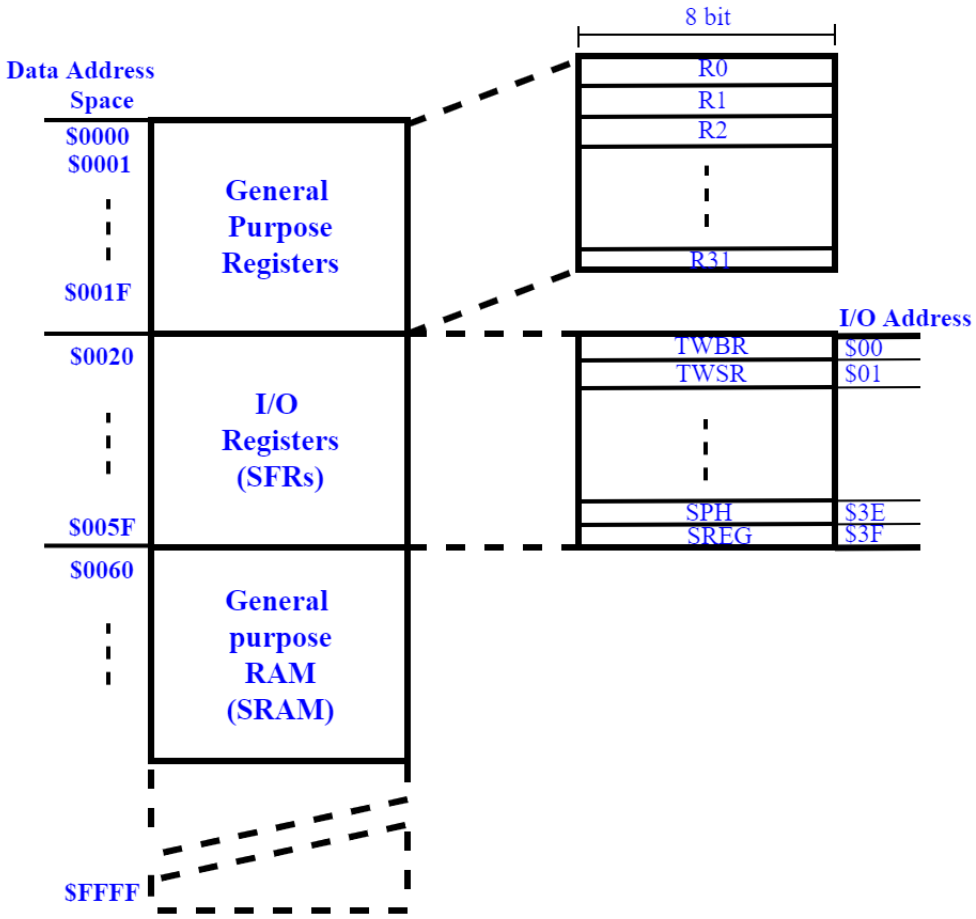

FIGURE 7.2: AVR SRAM

The GPRs use 32 bytes of data memory space. They always take the address location 0x00-0x1F in the data memory space, regardless of the AVR chip number. The I/O memory is dedicated to specific functions such as status register, timers, serial communication, I/O ports, ADC, and so on. The function of each I/O memory location is fixed by the CPU designer at the time of design because it is used for control of the microcontroller or peripherals.

The AVR I/O memory is made of 8-bit registers. The number of locations in the data memory set aside for I/O memory depends on the pin numbers and peripheral functions supported by that chip, although the number can vary from chip to chip even among members of the same family. However, all of the AVRs have at least 64 bytes of I/O memory locations. This 64-byte section is called standard I/O memory. In AVRs with more than 32 I/O pins (e.g., ATmega64, ATmega128, and ATmega256) there is also an extended I/O memory, which contains the registers for controlling the extra ports and the extra peripherals. In other microcontrollers, the I/O registers are called SFRs (special function registers) since each one is dedicated to a specific function. In contrast to SFRs, the GPRs do not have any specific function and are used for storing general data.

7.2 Parallel Ports

In the AVR family, there are many ports for I/O operations, depending on which family member you choose. For example, ATmega32 has a 40-pin chip. A total of 32 pins are set aside for the four ports PORTA, PORTB, PORTC, and PORTD. The rest of the pins are designated as VCC, GND, XTAL1, XTAL2, RESET, AREF, AGND, and AVCC.

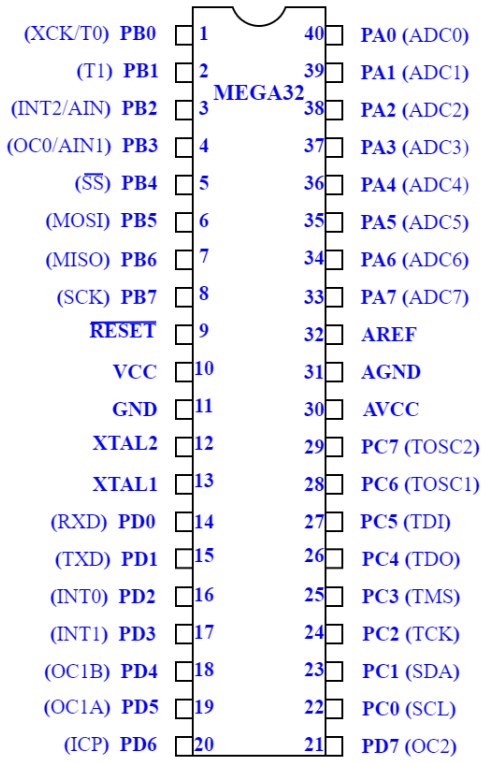

FIGURE 7.3: Atmega32

All ports have 8 pins. Each port has three I/O registers associated with it. They are designated as PORTx, DDRx, and PINx. For example, for Port B we have PORTB, DDRB, and PINB. Notice that DDR stands for Data Direction Register, and PIN stands for Port Input pins. Also, notice that each of the I/O registers is 8 bits wide, and each port has a maximum of 8 pins, therefore each bit of the I/O registers affects one of the pins.

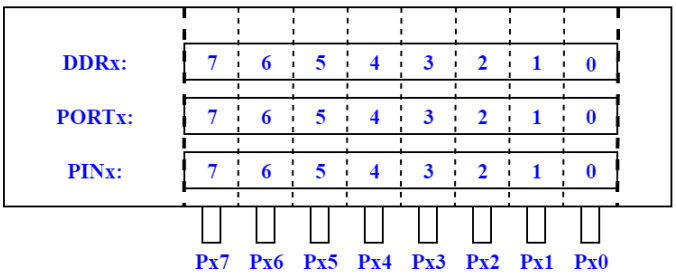

FIGURE 7.4: Port register

Each of the ports A - D in the ATmega32 can be used for input or output. The DDRx I/O register is used solely for the purpose of making a given port an input or output port. For example, to make a port an output, we write 1s to the DDRx register. In other words, to output data to all of the pins of the Port B, we must first put 0b11111111 into the DDRB register to make all of the pin's output. To make a port an input port, we must first put 0s into the DDRx register for that port, and then bring in (read) the data present at the pins.
As an aid for remembering that the port is input when the DDR bits are 0s, imagine a person who has 0 dollars. The person can only get money, not give it. Similarly, when DDR contains 0s, the port gets data.
Notice that upon reset, all ports have the value 0x00 in their DDR registers. This means that all ports are configured as an input. To read the data present at the pins, we should read the PIN register. It must be noted that to bring data into CPU from pins we read the contents of the PINx register, whereas to send data out to pins we use the PORTx register.

Need to remember

There is a pull-up resistor for each of the AVR pins. If we put 1s into bits of the PORTx register, the pull-up resistors are activated. In cases in which nothing is connected to the pin or the connected devices have high impedance, the resistor pulls up the pin. If we put 0s into the bits of the PORTx register, the pull-up resistor is inactive.

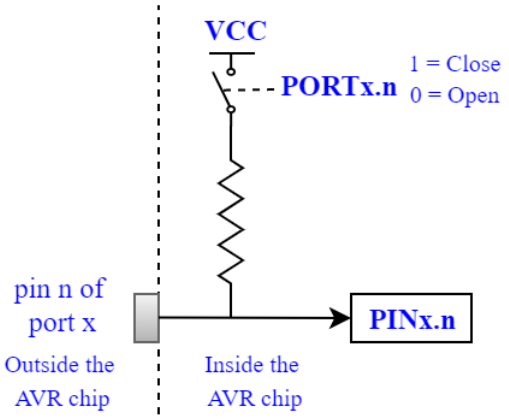

FIGURE 7.5: Port pull-up resister

The pins of the AVR microcontrollers can be in four different states according to the values of PORTx and DDRx.

PORTx \ DDRx.	0	1
0	Input and high impendance	Out 0
1	Input and pull-up	Out 1

7.3 Timers and Counters

Many applications need to count an event or generate time delays. So, there are counter registers in microcontrollers for this purpose. When we want to count an event, we connect the external event source to the clock pin of the counter register. Then, when an event occurs externally, the content of the counter is incremented, in this way, the content of the counter represents how many times an event has occurred. When we want to generate time delays, we connect the oscillator to the clock pin of the counter. So, when the oscillator ticks, the content of the counter is incremented. As a result, the content of the counter register represents how many ticks have occurred from the time we have cleared the counter. Since the speed of the oscillator in a microcontroller is known, we can calculate the tick period, and from the content of the counter register, we will know how much time has elapsed.

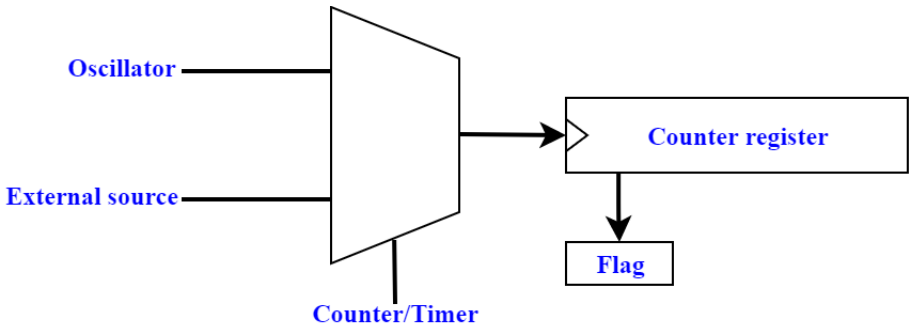

FIGURE 7.6: Timer, counter

One way to generate a time delay is to clear the counter at the start time and wait until the counter reaches a certain number. For example, consider a microcontroller with an oscillator with a frequency of 1 MHz, in the microcontroller, the content of the counter register increments once per microsecond. So, if we want a time delay of 100 microseconds, we should clear the counter and wait until it becomes equal to 100.
In the microcontrollers, there is a flag for each of the counters. The flag is set when the counter overflows, and it is cleared by software.
The second method to generate a time delay is to load the counter register and wait until the counter overflows and the flag is set.
For example, in a microcontroller with a frequency of 1 MHz, with an 8-bit counter register, if we want a time delay of 3 microseconds, we can load the counter register with 0xFD and wait until the flag is set after 3 ticks. After the first tick, the content of the register increments to 0xFE, after the second tick, it becomes 0xFF, and after the third tick, it overflows (the content of the register becomes 0x00) and the flag is set.
In ATmega32, there are three timers: Timer0, Timer1, and Timer2. Timer0 and Timer2

are 8-bit, while Timer1 is 16-bit. In this chapter we cover Timer0 as an 8-bit timer. If you learn to use the timers of ATmega32, you can easily use the timers of other AVRs.
Every timer needs a clock pulse to tick. The clock source can be internal or external. If we use the internal clock source, then the frequency of the crystal oscillator is fed into the timer. Therefore, it is used for time delay generation and consequently is called a **timer**. By choosing the external clock option, we feed pulses through one of the AVR's pins. This is called a **counter**.

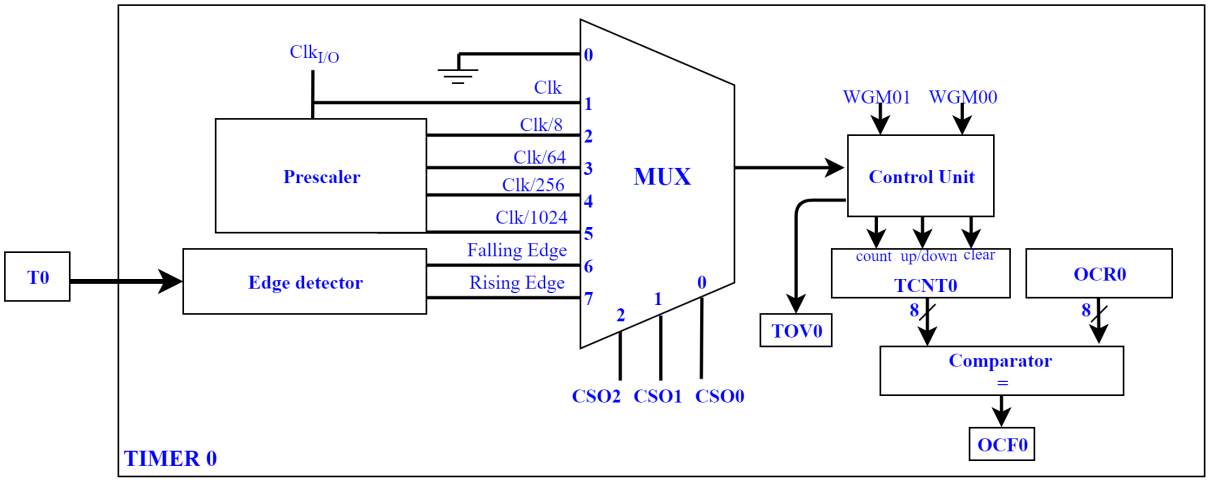

FIGURE 7.7: Timer counter registers

AVR, for each of the timers, there is a TCNTn (timer/counter) register. That means in ATmega32 we have TCNT0, TCNTI, and TCNT2. The TCNTn register is a counter. Upon reset, the TCNTn contains zero. It counts up with each pulse.
The contents of the timers/counters can be accessed using the TCNTn. You can load a value into the TCNTn register or read its value.
Each timer has a TOVn (Timer Overflow) flag, as well. When a timer overflows, its TOVn flag will be set. Each timer also has the TCCRn (timer/counter control register) register for setting modes of operation.
For example, you can specify Timer0 to work as a timer or a counter by loading proper values into the TCCR0. Each timer also has an OCRn (Output Compare Register) register. The content of the OCRn is compared with the content of the TCNTn. When they are equal the OCFn (Output Compare Flag) flag will be set. The timer registers are located in the I/O register memory.

TCNT0 register

D7	D6	D5	D4	D3	D2	D1	D0

TCCR0 register

FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

FOC0 -Force compare match. This is a write-only bit, which can be used while generating a wave. Writing 1 to it causes the wave generator to act as if a compare match had occurred.

WGM00, WGM01 - Timer0 mode selector bits

0 0 - Normal

0 1 - Clear Timer on Compare Match

1 0 - PWM, phase correct

1 1 - Fast PWM

COM01, COM00 - Compare Output Mode

0 0 - Normal port operation, OC0 disconnected

0 1 - Toggle OC0 on compare match

1 0 - Clear OC0 on compare match

1 1 - Ser OC0 on compare match

CS02, CS01, CS00 - Timer0 clock selector

0 0 0 - No clock source, T/C stopped

0 0 1 - Clk, No prescaling

0 1 0 - Clk/8

0 1 1 - Clk/64

1 0 0 - Clk/256

1 0 1 - Clk/1024

1 1 0 - External clk source on T0 pin. Clk falling edge

1 1 1 - External clk source on T0 pin. Clk rising edge

The **TIFR** register contains the flags of different timers.

OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0

TOV0 - Timer0 overflow flag bit. 0- did not overflow, 1- has overflow

OCF0 - Timer0 output compare flag bit. 0 - compare match didn't occur, 1 compare match occur

TOV1 - Timer1 overlow flag bit

OCF1B - Timer1 output compare B match flag

OCF1A - Timer1 output compare A match flag

ICF1 - Input capture flag

TOV2 - Timer2 overflow flag
OCF2 - Timer2 output compare match flag

Timer0 can work in four different modes: Normal, phase correct PWM, CTC, and Fast PWM. The WGM01 and WGM00 bits are used to choose one of them.
In **Normal mode**, the content of the timer/counter increments with each clock. It counts up until it reaches its max of 0xFF. When it rolls over from 0xFF to 0x00, it sets high a flag bit called TOV0 (Timer Overflow). This timer flag can be monitored.

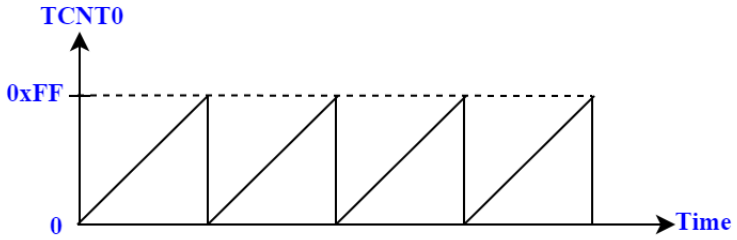

FIGURE 7.8: Timer normal mode

7.3.1 Pulse Width Modulation

We learned how to use AVR timers to generate delay and count external events. AVR timers have other features as well. They can be used for generating different square waves or capturing events and measuring the frequency and duty cycle of waves.
For each timer there is, at least, an OCRn register (like OCR0 for Timer0). The value of this register is constantly compared with the TCNTn register, and when a match occurs, the OCFn flag will be set to high. In each AVR timer there is a waveform generator. The waveform generator can generate waves on the OCn pin.
The WGMn and COMn bits of the TCCR register determines how the waveform generator works. When the TCNTn register reaches Top or Bottom or compare match occurs, the waveform generator is informed. Then the waveform generator changes the state of the OC0 pin according to the mode of the timer (WGM01:00 bits of the TCCR0 register) and the COM01 (Compare Output Mode) and COM00 bits. In ATmega32, OC0 is the alternative function of PB3. In other words, the PB3 functions as an I/O port when both COM01 and COM00 are zero. Otherwise, the pin acts as a wave generator pin controlled by a waveform generator. Since the DDR register represents the direction of the I/O pin, we should set the OC0 pin as an output pin when we want to use it for generating waves.

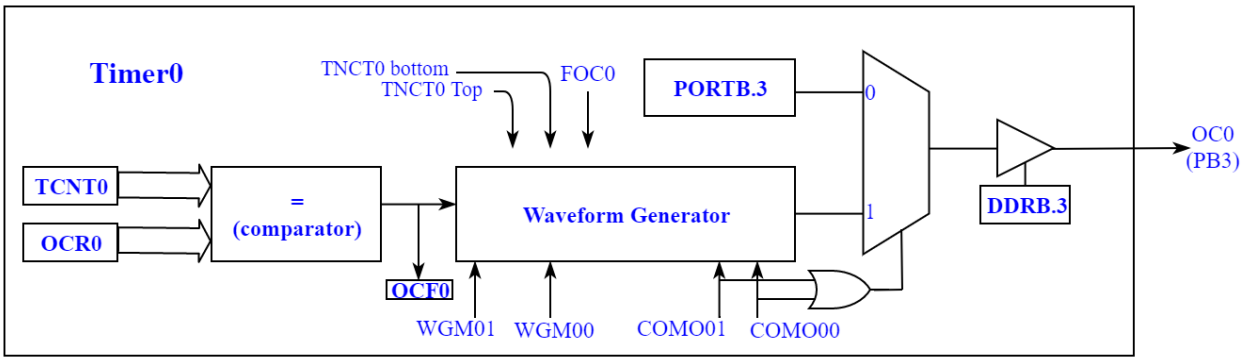

FIGURE 7.9: Wave generator

Now we discuss the PWM feature of the AVR. The ATmega32 comes with three timers, which can be used as wave generators. PWM is so widely used in DC motor control that some microcontrollers come with the PWM circuitry embedded in the chip.

Need to remember

In such microcontrollers all we have to do is load the proper registers with the values of the high and low portions of the desired pulse, and the rest is taken care of by the microcontroller. This allows the microcontroller to do other things.
The ability to control the speed of the DC motor using PWM is one reason that DC motors are often preferred over AC motors. For a given fixed load we can maintain a steady speed by using a method called pulse width modulation (PWM). By changing (modulating) the width of the pulse applied to the DC motor we can increase or decrease the amount of power provided to the motor, thereby increasing or decreasing the motor speed. Notice that, although the voltage has a fixed amplitude, it has a variable duty cycle. That means the wider the pulse, the higher the speed.

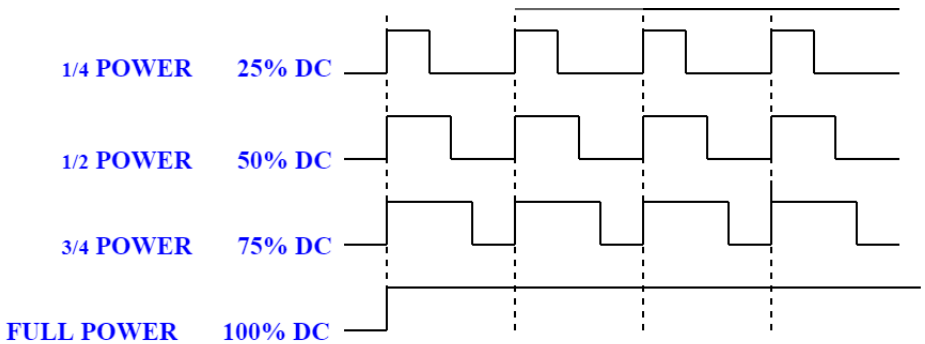

FIGURE 7.10: PWM signal

In the **Fast PWM mode**, the counter counts as it does in the Normal mode. After the timer is started, it starts to count up.

Important

It counts up until it reaches its limit of 0xFF. When it rolls over from 0xFF to 00, it sets HIGH the TOVO flag. The Figure7.11 shows the reaction of the waveform generator when compare match occurs while the timer is in Fast PWM mode.

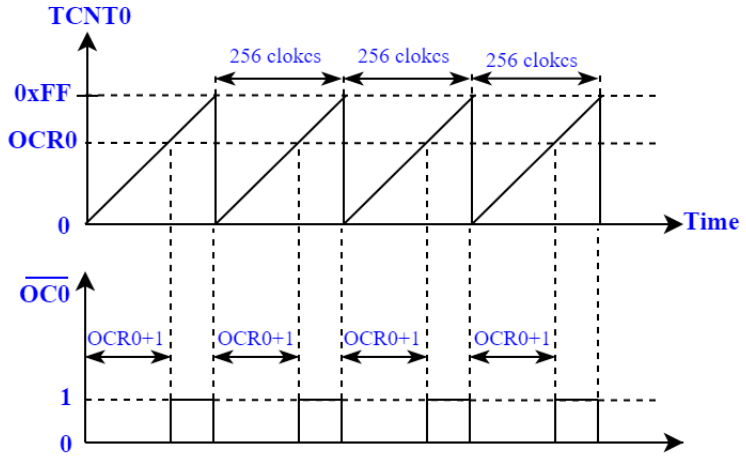

FIGURE 7.11: PWM mode

In Fast PWM mode, the timer counts from 0 to top (0xFF in 8-bit counters) and then rolls over. So, the frequency of the generated wave is 1/256 of the frequency of timer clock. The frequency of the timer clock can be selected using the prescaler. So, in 8-bit timers the frequency of the generated wave can be calculated as follows (N is determined by the prescaler):

$$F_{generated.wave} = \frac{F_{timer.clock}}{256} \tag{7.1}$$

$$F_{timer.clock} = \frac{F_{oscillator}}{N} \tag{7.2}$$

$$F_{generated.wave} = \frac{F_{oscillator}}{256 * N} \tag{7.3}$$

7.4 Interrupts

A single microcontroller can serve several devices. There are two methods by which devices receive service from the microcontroller: **interrupts** or **polling**. In the interrupt method, whenever any device needs the microcontroller's service, the device notifies it by sending an interrupt signal. Upon receiving an interrupt signal, the microcontroller stops whatever it is doing and serves the device. The program associated with the interrupt is called the **interrupt service routine** (ISR) or interrupt handler.
In polling, the microcontroller continuously monitors the status of a given device, when the status condition is met, it performs the service. After that, it moves on to monitor the next device until each one is serviced. Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the microcontroller. The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time, of course), each device can get the attention of the microcontroller based on the priority assigned to it. The polling method cannot assign priority because it checks all devices in a round-robin fashion. More importantly, in the interrupt method the microcontroller can also ignore (mask) a device request for service. This also is not possible with the polling method.

Important

The most important reason that the interrupt method is preferable is that the polling method wastes much of the microcontroller's time by polling devices that do not need service. So interrupts are used to avoid tying down the microcontroller. That is a waste of microcontroller time that could have been used to perform some useful tasks.
In the case of the timer, if we use the interrupt method, the microcontroller can go about doing other tasks, and when the TOV0 flag is raised, the timer will interrupt the microcontroller in whatever it is doing. For every interrupt, there must be an interrupt service routine (ISR) or interrupt handler. When an interrupt is invoked, the microcontroller runs the interrupt service routine. Generally, in most microprocessors, for every interrupt, there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector table.

Interrupt	ROM Location (Hex)
Reset	0000
External Interrupt request 0	0002
External Interrupt request 1	0004
External Interrupt request 2	0006
Timer/Counter2 Compare Match	0008
Timer/Counter2 Overflow	000A
Timer/Counter1 Capture Event	000C
Timer/Counter1 Compare Match A	000E
Timer/Counter1 Compare Match B	0010
Timer/Counter1 Overflow	0012
Timer/Counter0 Compare Match	0014
Timer/Counter0 Overflow	0016
SPI Transfer complete	0018
USART, Receive Complete	001A
USART, Data Register Empty	001C
USART, Transmit Complete	001E
ADC Conversion Complete	0020
EEPROM Ready	0022
Analog Comparator	0024
I2C	0026
Store Program Memory Ready	0028

Need to remember

Upon activation of an interrupt, the microcontroller goes through the following steps:

1. It finishes the instruction it is currently executing and saves the address of the next instruction (program counter) on the stack.

2. It jumps to a fixed location in memory called the interrupt vector table. The interrupt vector table directs the microcontroller to the address of the interrupt service routine (ISR).

3. The microcontroller starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETI (return from interrupt).

4. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then it starts to execute from that address.

Important

Notice from Step 4 the critical role of the stack. For this reason, we must be careful in manipulating the stack contents in the ISR. Specifically, in the ISR, just as in any CALL subroutine, the number of pushes and pops must be equal. Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated. The interrupts must be enabled (unmasked) by software in order for the microcontroller to respond to them. The D7 bit of the SREG (Status Register) register is responsible for enabling and disabling the interrupts globally. The I bit makes the job of disabling all the interrupts easy. Bit D7 (I) of the SREG register must be set to HIGH to allow the interrupts to happen. This is done with the "SEI" (Set Interrupt) instruction.

There are three external hardware interrupts in the ATmega32: INT0, INT1, and INT2. They are located on pins PD2, PD3, and PB2, respectively. As we saw in Table 10-1, the interrupt vector table locations 0x02, 0x04, and 0x06 are set aside for INT0, INT1, and INT2, respectively. The hardware interrupts must be enabled before they can take effect. This is done using the INTx bit located in the **GICR** register.

INT1	INT0	INT2	-	-	-	IVSEL	IVCE

INT0 - External Interrupt Request 0 Enable =1
INT1 - External Interrupt Request 1 Enable =1
INT2 - External Interrupt Request 2 Enable =1

The INT0 is a low-level-triggered interrupt by default, which means, when a low signal is applied to pin PD2 (PORTD.2), the controller will be interrupted and jump to location 0x0002 in the vector table to service the ISR.

7.5 Analog to Digital Convertor

Analog-to-digital converters are among the most widely used devices for data acquisition. Digital computers use binary (discrete) values, but in the physical world everything is analog (continuous). Temperature, pressure (wind or liquid), humidity, and velocity are a few examples of physical quantities that we deal with every day. A physical quantity is converted to electrical (voltage, current) signals using a device called a transducer. Transducers are also referred to as sensors. Sensors for temperature, velocity, pressure, light, and many other natural quantities produce an output that is voltage (or current). Therefore, we need an analog-to-digital converter to translate the analog signals to digital numbers so that the microcontroller can read and process them.

The ADC has an **n-bit resolution**, where n can be 8, 10, 12, 16, or even 24 bits. Higher-resolution ADCs provide a smaller step size, where the step size is the smallest change that can be discerned by an ADC. Although the resolution of an ADC chip is decided at the time of its design and cannot be changed, we can control the step size with the help of what is called Vref. Vref is an input voltage used for the reference voltage. The voltage connected to this pin, along with the resolution of the ADC chip, dictates the step size. For an 8-bit ADC, the step size is Vref/256 because it is an 8-bit ADC, and 2 to the power of 8 gives us 256 steps. For example, if the analog input range needs to be 0 to 4 volts, Vref is connected to 4 volts.
That gives 4 V/256 = 15.62 mV for the step size of an 8-bit ADC.

In addition to resolution, the **conversion time** is another major factor in judging an ADC. Conversion time is defined as the time it takes the ADC to convert the analog input to a digital (binary) number. The conversion time is dictated by the clock source connected to the ADC in addition to the method used for data conversion and technology used in the fabrication of the ADC chip such as MOS or TTL technology.
The ADC peripheral of the ATmega32 has the following characteristics:

(a) It is a 10-bit ADC.

(b) It has 8 analog input channels, 7 differential input channels, and 2 differential input channels with an optional gain of 10x and 200x.

(c) The converted output binary data is held by two special function registers called ADCL (A/D Result Low) and ADCH (A/D Result High).

(d) Because the ADCH:ADCL registers give us 16 bits and the ADC data out is only 10 bits wide, 6 bits of the 16 are unused. We have the option of making either the upper 6 bits or the lower 6 bits unused.

(e) We have three options for Vref. Vref can be connected to AVCC (Analog Vcc), internal 2.56V reference, or external AREF pin. The conversion time is dictated by the

crystal frequency connected to the XTAL pins (Fosc) and ADPSO:2 bits.

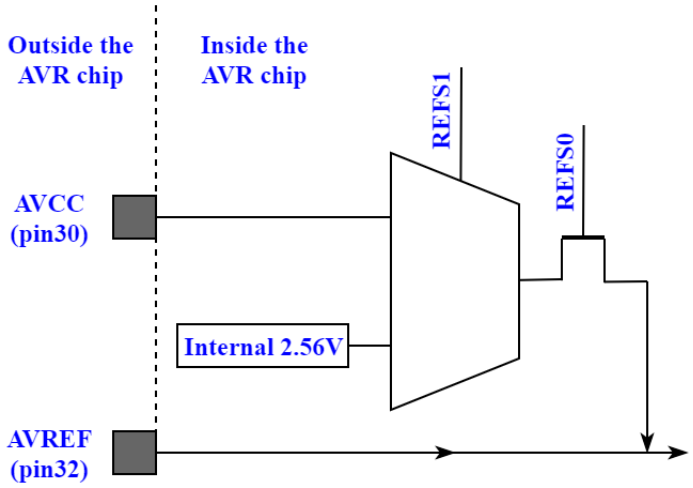

FIGURE 7.12: ADC

In the AVR microcontroller, five major registers are associated with the ADC. They are ADCH (high data), ADCL (low data), ADCSRA (ADC Control and Status Register), ADMUX (ADC multiplexer selection register), and SPIOR (Special Function I/O Register).

ADMUX register

REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0

REFS1	REFS0	Vref
0	0	AREF pin
0	1	AVCC pin
1	0	Reserved
1	1	Internal 2.56V

ADLAR - ADC Left Adjust Result

MUX4-0	Single ended Input
00000	ADC0
00001	ADC1
00010	ADC2
00011	ADC3
00100	ADC4
00101	ADC5
00110	ADC6
00111	ADC7

ADCSRA register

ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

ADEN - ADC Enable
ADSC - ADC Start Conversion
ADATE - ADC Auto Trigger Enable
ADIF - ADC Interrupt Flag
ADIE - ADC Interrupt Enable
ADPS2:0 ADC Prescaler Select Bits

ADPS2	ADPS1	ADPS0	ADC Clock
0	0	0	Reserved
0	0	1	CLK/2
0	1	0	CLK/4
0	1	1	CLK/8
1	0	0	CLK/16
1	0	1	CLK/32
1	1	0	CLK/64
1	1	1	CLK/128

7.6 Serial Communication Interfaces

Computers transfer data in two ways: parallel and serial. In parallel data transfers, often eight or more lines (wire conductors) are used to transfer data to a device that is only a few feet away. Devices that use parallel transfers include printers and IDE hard disks, each uses cables with many wires. Although a lot of data can be transferred in a short amount of time by using many wires in parallel, the distance cannot be great. To transfer to a device located many meters away, the serial method is used.
In serial communication, the data is sent one bit at a time, in contrast to parallel communication, in which the data is sent a byte or more at a time. The AVR has serial communication capability built into it, thereby making possible fast data transfer using only a few wires. For these reasons, serial communication is used for transferring data between two systems located at distances of hundreds of feet to millions of miles apart.

In data transmission, if the data can be both transmitted and received, it is a duplex transmission. This is in contrast to simplex transmissions such as with printers, in which the computer only sends data. Duplex transmissions can be half or full duplex, depending on whether or not the data transfer can be simultaneous. If data is transmitted one way

at a time, it is referred to as **half-duplex**. If the data can go both ways at the same time, it is **full-duplex**. Of course, full-duplex requires two wire conductors for the data lines (in addition to the signal ground), one for transmission and one for a reception, in order to transfer and receive data simultaneously.

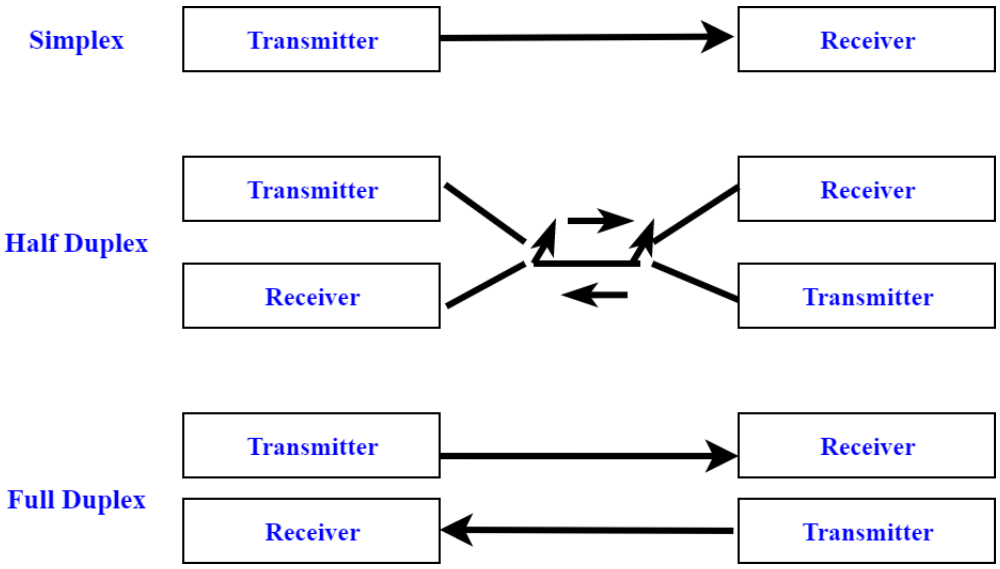

FIGURE 7.13: Serial connection

7.6.1 USART

Serial data communication uses two methods, **asynchronous** and **synchronous**. The synchronous method transfers a block of data (characters) at a time, whereas the asynchronous method transfers a single byte at a time. These chips are commonly referred to as UART (universal asynchronous receiver-transmitter) and USART (Universal Aynchronous-Asynchronous Receiver-Transmitter). The AVR chip has a built-in USART. In the asynchronous method, each character is placed between *start* and *stop* bits. This is called framing. In data framing for asynchronous communications, the data, such as ASCII characters, are packed between a start bit and a stop bit. The start bit is always one bit, but the stop bit can be one or two bits. The start bit is always a 0 (low), and the stop bit(s) is 1 (high). For example, the ASCII character "A" (8-bit binary 0100 0001) is framed between the start bit and a single stop bit. Notice that the LSB is sent out first. The transmission begins with a start bit (space) followed by D0, the LSB, then the rest of the bits until the MSB (D7), and finally, the one stop bit indicating the end of the character "A".

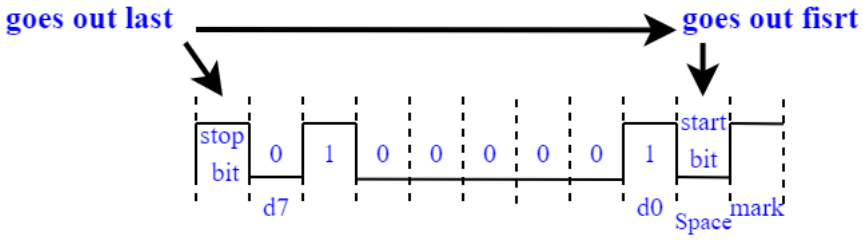

FIGURE 7.14: Asynchronous data transfer

In some systems, the *parity* bit of the character byte is included in the data frame in order to maintain data integrity. This means that for each character (7- or 8-bit, depending on the system) we have a single parity bit, in addition, to start and stop bits. The parity bit is odd or even. In the case of an odd parity bit the number of 1s in the data bits, including the parity bit, is odd. Similarly, in an even parity bit system the total number of bits, including the parity bit, is even. For example, the ASCII character "A", binary 0100 0001, has 0 for the even parity bit. UART chips allow programming of the parity bit for odd, even, and no-parity options.

The rate of data transfer in serial data communication is stated in bps (bits per second). Another widely used terminology for bps is the **baud rate**. The data transfer rate of a given computer system depends on communication ports incorporated into that system. For example, the early IBM PC/XT could transfer data at the rate of 100 to 9600 bps. In recent years, however, Pentium-based PCs transfer data at rates as high as 56K. Notice that in asynchronous serial data communication, the baud rate is generally limited to 115,200 bps. The ATmega32 has two pins that are used specifically for transferring and receiving data serially. These two pins are called TX and RX and are part of the Port D group (PD0 and PD1) of the 40-pin package. Pin 15 of the ATmega32 is assigned to TX and pin 14 is designated as RX. These pins are TTL compatible.

In the AVR microcontroller, five registers are associated with the USART. They are UDR (USART Data Register), UCSRA, UCSRB, UCSRC (USART Control Status Register), and UBRR (USART Baud Rate Register). We examine each of them and show how they are used in full-duplex serial data communication. The AVR transfers and receives data serially at many different baud rates. The baud rate in the AVR is programmable. This is done with the help of the 8-bit register called UBRR. For a given crystal frequency, the value loaded into the UBRR decides the baud rate.

The relation between the value loaded into UBBR and the Fosc (frequency of oscillator connected to the XTALI and XTAL2 pins) is dictated by the following formula:

$$DesiredBaudRate = \frac{F_{osc}}{16(X+1)} \tag{7.4}$$

where X is the value we load into the UBRR register.

$$X = (\frac{F_{osc}}{16(DesiredBaudRate)}) - 1 \tag{7.5}$$

To get the X value for different baud rates we can solve the equation as follows:

Baud Rate	UBRR(Hex)
38400	C
19200	19
9600	33
4800	37
2400	CF
1200	19F

In the AVR, to provide full-duplex serial communication, there are two shift registers referred to as Transmit Shift Register and Receive Shift Register. Each shift register has a buffer that is connected to it directly. These buffers are called Transmit Data Buffer Register and Receive Data Buffer Register. The USART Transmit Data Buffer Register and USART Receive Data Buffer Register share the same I/O address, which is called USART Data Register or UDR. When you write data to UDR, it will be transferred to the Transmit Data Buffer Register (TXB), and when you read data from UDR, it will return the contents of the Receive Data Buffer Register (RXB).

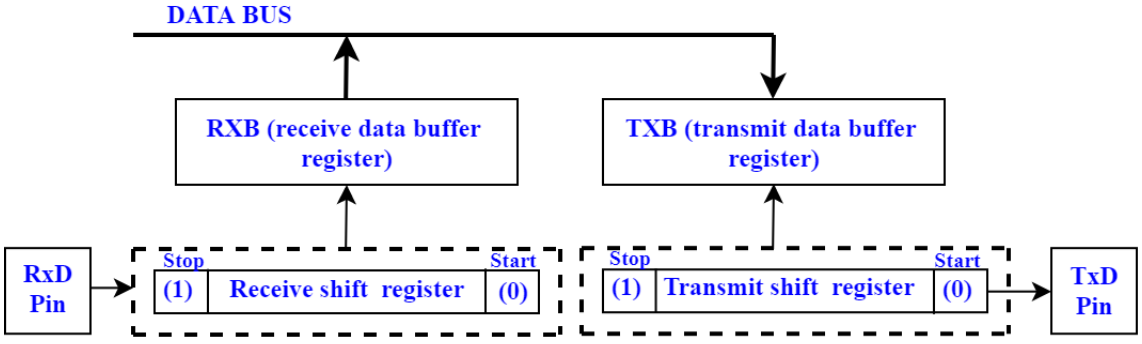

FIGURE 7.15: UDR register

UCSRs are 8-bit control registers used for controlling serial communication in the AVR. There are three USART Control Status Registers in the AVR. They are UCSRA, UCSRB, and UCSRC.

UCSRA register

RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM

RXC - USART Receive Complete. This flag bit is set when there are new data in the receive buffer that are not read yet.
TXC - USART Transmit Complete. This flag bit is set when the entire frame in the transmit shift register has been transmitted and there are no new data available in the transmit data buffer register.
UDRE - USART Data Register Empty. This flag is set when the transmit data buffer is empty and it is ready to receive new data.
FE - Frame Error. This bit is set if a frame error has occurred in receiving the next character in the receive buffer.
DOR - Data OverRun. This bit is set if a data overrun is detected.
PE - Parity Error.
U2X - Double the USART Transmission Speed
MPCM - Multiprocessor Communication Mode

UCSRB register

RXCIE	TXCIE	UDRIE	RXEN	TXEN	UCSZ2	RXB8	TXB8

RXCIE - Receive Complete Interrupt Enable
TXCIE - Transmit Complete Interrupt Enable
UDRIE - USART Data Register Empty Interrupt Enable
RXEN - Receive Enable. To enable the USART receiver you should set this bit to one
TXEN - Transmit Enable. To enable the USART transmitter you should set this bit to one
UCSZ2 -Character Size
RXB8 - Receive data bit 8
TXB8 - Transmit data bit 8

UCSRC register

URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL

URSEL - Register Select. This bit selects to access either UCSRC or the UBRRH register
UMSEL - USART Mode Select
0 - Asynchronous operation
1 - Synchronous operation
UPM1:0 - Parity Mode
00 = Disabled

01 = Reserved
10 = Even Parity
11 = Odd Parity
USBS - Stop Bit Select
0 = 1 bit
1 = 2 bit
UCSZ1:0 - Character Size
UCPOL - Clock Polarity for synchronous mode

UCSZ2	UCSZ1	UCSZ0	Character Size
0	0	0	5
0	0	1	6
0	1	0	7
0	1	1	8
1	1	1	9

Review questions

1. What kind of memories has ATmega32?
2. What is the difference between SRAM and EEPROM?
3. How work parallel port? How many parallel ports have ATmega32?
4. How many registers associated with each port? Name it.
5. What is pull up resister? How to control it using the software?
6. What is the main difference between timer and counter?
7. If TCCR0 =0x07 is it timer or counter?
8. Draw timer Normal mode time diagram.
9. Draw the PWM signal with 75 percent of the duty cycle.
10. What is the difference between polling and interrupt services?
11. Explain how is interrupt is executed.
12. Which of the following ADC sizes provides the best resolution?

a. 8-bit
b. 10-bit
c. 12-bit
d. they are all the same

13. Calculate the step size of 10-bit ADC if Vref is 5V
14. What is the difference between asynchronous and synchronous serial communications?
15. Draw a timing diagram of 0x79 asynchronous data transfer.

CHAPTER 8

Introduction to PID controller

Topic 8

PID controller

- Proportional–Integral–Derivative
- PID formulas
- PID tunnig
- Implementation PID controller

The **Proportional–Integral–Derivative** (PID) controller is often referred to as a 'three-term' controller[1]. It is currently one of the most frequently used controllers in the process industry. In a PID controller, the control variable is generated from a term proportional to the error, a term which is the integral of the error, and a term which is the derivative of the error.

• **Proportional**: the error is multiplied by a gain K_p. A very high gain may cause instability, and a very low gain may cause the system to drift away.

• **Integral**: the integral of the error is taken and multiplied by a gain K_i. The gain can be adjusted to drive the error to zero in the required time. A too high gain may cause oscillations and a too low gain may result in a sluggish response.

• **Derivative**: The derivative of the error is multiplied by a gain K_d. A gain, if the gain is too high the system may oscillate and if the gain is too low the response may be sluggish

The block diagram of the classical continuous-time PID controller.

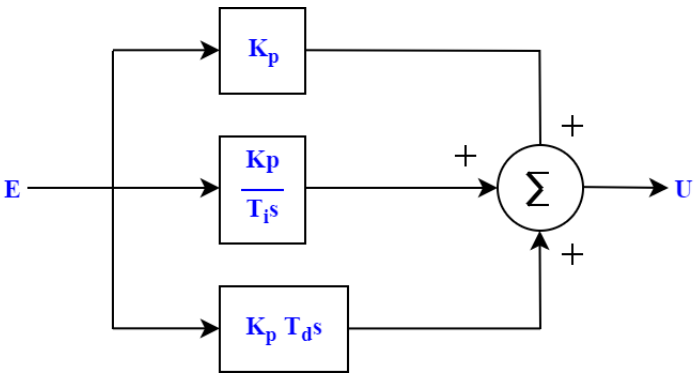

FIGURE 8.1: PID control

Tuning the controller involves adjusting the parameters K_p, K_d and K_i in order to obtain a satisfactory response. The characteristics of PID controllers are well known

and well established, and most modern controllers are based on some form of PID. The input-output relationship of a PID controller can be expressed as

$$u(t) = K_p \left[e(t) + \frac{1}{T_i} \int_0^t e(t)dt + T_d \frac{de(t)}{dt} \right] \tag{8.1}$$

Where $u(t)$ is the output from the controller and $e(t) = r(t) - y(t)$, in which $r(t)$ is the desired set-point (reference input) and $y(t)$ is the plant output. Ti and Td are known as the integral and derivative action time, respectively.

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} + u_0 \tag{8.2}$$

$$K_i = \frac{K_p}{T_i} \tag{8.3}$$

and

$$K_d = K_p T_d \tag{8.4}$$

8.1 PID tuning

When a PID controller is used in a system it is important to tune the controller to give the required response. Tuning a PID controller involves selecting values for the controller parameters K_p, T_i and T_d. There are many techniques for tuning a controller, ranging from the first techniques described by J.G. Ziegler and N.B. Nichols (known as the Ziegler–Nichols tuning algorithm) in 1942 and 1943, to recent auto-tuning controllers. Ziegler and Nichols suggested values for the PID parameters of a plant based on open-loop or closed-loop tests of the plant. According to Ziegler and Nichols, the open-loop transfer function of a system can be approximated with a time delay and a single-order system.

8.2 Implementation PID controller

To implement the PID controller using a digital computer we have to convert from a continuous to a discrete representation. There are several methods for doing this and the simplest is to use the trapezoidal approximation for the integral and the backward difference approximation for the derivative:

$$\frac{de(t)}{dt} \approx \frac{e(kT) - e(kT - T)}{T} \tag{8.5}$$

and

$$\int_0^t e(t)dt \approx \sum_{k=1}^{n} Te(kT) \tag{8.6}$$

$$u(kT) = K_p \left[e(kT) + T_d \frac{e(kT) - e(kT-T)}{T} + \frac{T}{T_i} \sum_{k=1}^{n} e(kT) \right] + u_0 \tag{8.7}$$

The PID given above is now in a suitable form that can be implemented on a digital computer. This form of the PID controller is also known as the positional PID controller. Notice that new control action is implemented at every sample time.

Review questions

1. Explain the influence of Proportional control on the system.
2. Explain the influence of Integral control on the system.
3. Explain the influence of Derivative control on the system.
4. Draw the PID control block diagram of motor control.
5. Write the equation of PID control.
6. How is tunned PID control?
7. How to implement PID control into the digital system?
8. Write a C code of P control.
9. Write a C code of PD control.
10. Write a C code of PID control.

CHAPTER 9

Controller based projects

9.1 Laboratory work 1

Wolf scaring device

Objectives of laboratory work

- Understanding the parallel port as output
- Developing practical skills
- Learn to program parallel ports
- Understanding registers of parallel ports

Laboratory assignments

Using ATmega32 connected with 2 buttons and with one LED, write a program that blinks the LEDs in random sequences and frequencies. Imagine that it is an open-loop system, and LEDs are placed in the yard that will scare wolves.

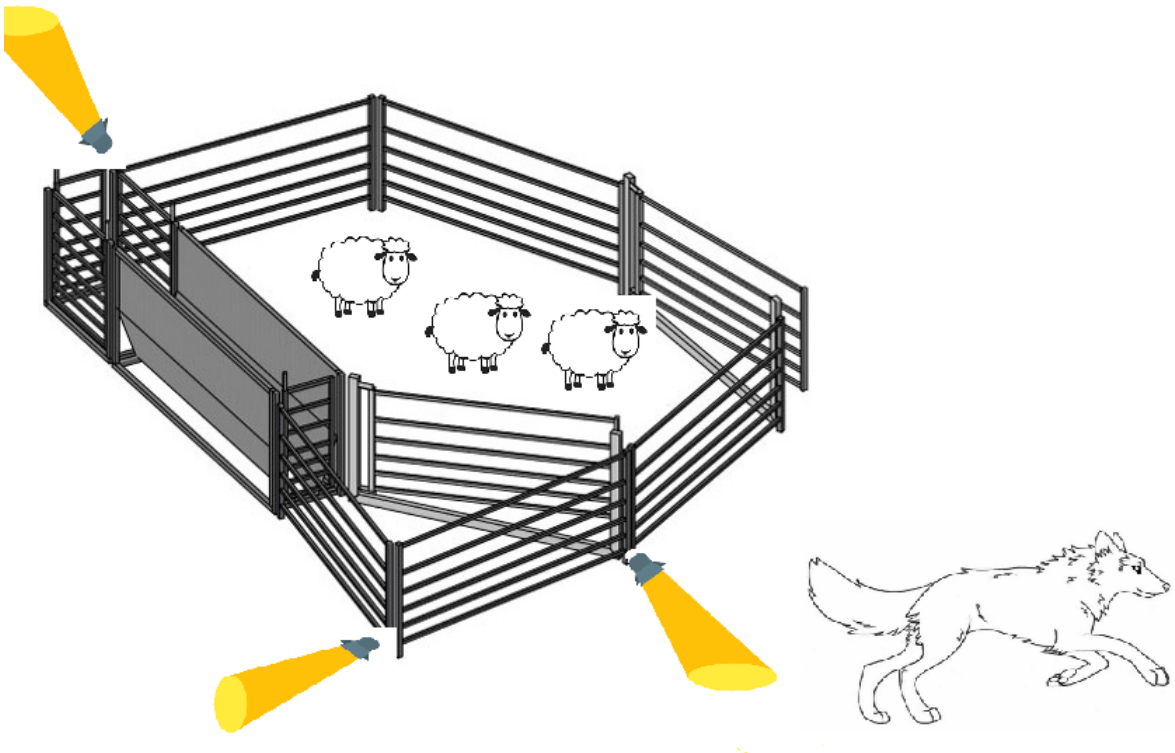

FIGURE 9.1: Wolf scaring device

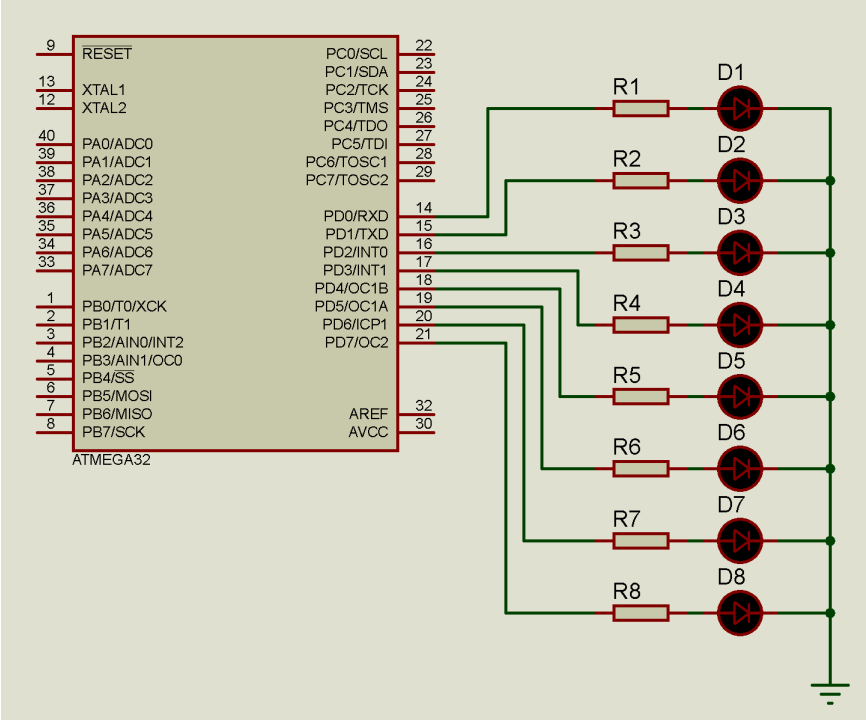

FIGURE 9.2: Device schematic

Example 9.1.1

Write an AVR C program to send values 00—FF to Port B.

Solution:

```
#include <avr/io.h>          //standard AVR header
int main (void)
{
unsigned char z;
    DDRB = 0xFF;
    for (z = 0; z <= 255; z++)
        PORTB = z;             //PORTB is output
    return 0;
)
```

Example 9.1.2

Write an AVR C program to toggle all the bits of Port B 200 times.

Solution:

```
#include <avr/io.h>          //standard AVR header
int main (void)              //the code starts from here
{
DDRB = 0xFF;                 //PORTB is output
PORTB = 0xAA;                //PORTS is 10101010
unsigned char z;
for(z=0; z < 200; z++)       //run the next line 200 times
    PORTB = ~ PORTB;         //toggle PORTB
while(1);                    //stay here forever
return 0;
}
```

Example 9.1.3

Write ma AVR C program to send values of —4 to +4 to Port B.

Solution:

```
#include <avr/io.h>          //standard AVR header
int main(void)
{
char mynun()= {-4,-3,-2,-1,0,1,2,3,4}
   unsigned char z;
DDRB = 0xFF;                  //PORTB is output
for (z=0; z<=8; z++)
     PORTB = mynun( z) ;
while(1);                     //stay here forever
return 0;
}
```

Homework

1. Write a program of Wolf scarring device
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.2 Laboratory work 2

Traffic light control

Objectives of laboratory work

- Understanding the parallel port as output
- Developing practical skills
- Learn to program parallel ports
- Learn to write time delay using statements

Laboratory assignments

Using ATmega32 connected with 6 LEDs, write a program of the traffic light. Imagine that it is an open-loop system, and LEDs are placed in a crosswalk. Using statement get time delay with different times. For example, the red light will flash 20 seconds, yellow light 3 seconds and green light 15 seconds.

FIGURE 9.3: Traffic light

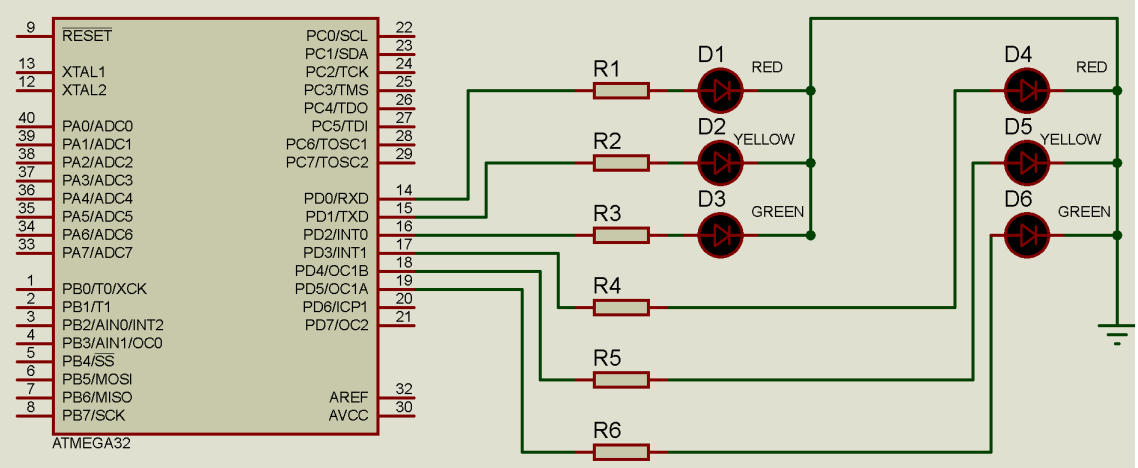

FIGURE 9.4: Traffic light schematic

Example 9.2.1

Write an AVR C program to toggle all the bits of Port B continuously with a 100 ms delay. Assume that the system is ATmega32 with XTAL = 8 MHz.

Solution:

```
#include <avr/io.h>                    //standard AVR header
void delay100ms()
{
    unsigned int i;
    for(i=0; i<42150; i++);            //try different numbers on your
}                                      //compiler and examine the result.
int main(void)
{
    DDRB = 0xFF;                       //PORTB is output
    while (1)
    {
        PORTB = 0xAA;
        delay100ms();
        PORTB = 0x55;
        delay100ms();
    }
    return 0;
}
```

Example 9.2.2

Write an AVR C program to toggle all the pins of Port C continuously with a 10 ms delay. Use a predefined delay function in Win AVR.

Solution:

```
#include <util/delay.h>          //delay loop functions
#include <avr/io.h>              //standard AVR header
int main(void)
{
    DDRB = 0xFF;                 //PORTB is output
    while (1)
    {
        PORTB = 0xFF;
        _delay_ms(10);
        PORTB = 0x55;
        _delay_ms(10);
    }
    return 0;
}
```

Homework

1. Write a program of Traffic light control
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.3 Laboratory work 3

Road Automatic Speed Bump

Objectives of laboratory work

- Understanding the parallel port as input
- Developing practical skills
- Learn to program parallel ports
- Learn to connect switch, buttons to parallel ports
- Understanding the work of pull-up resistor

Laboratory assignments

Using ATmega32 connected with 2 buttons and one led (motor). Imagine that these two buttons are sensors and used to measure the speed of the car. If button 1 is pressed then start counting until button 2 is pressed. If the counted value is small from some value that's mean that the car is going very fast and led will light up (Speed bump goes up). If the counted value is large that's mean that the car is going slow and light would not flash.

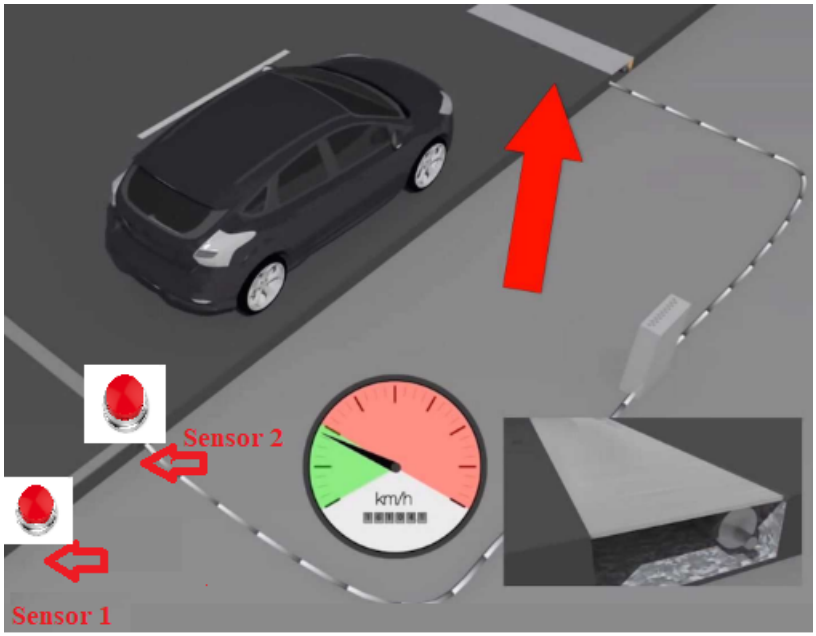

FIGURE 9.5: Road automatic bump

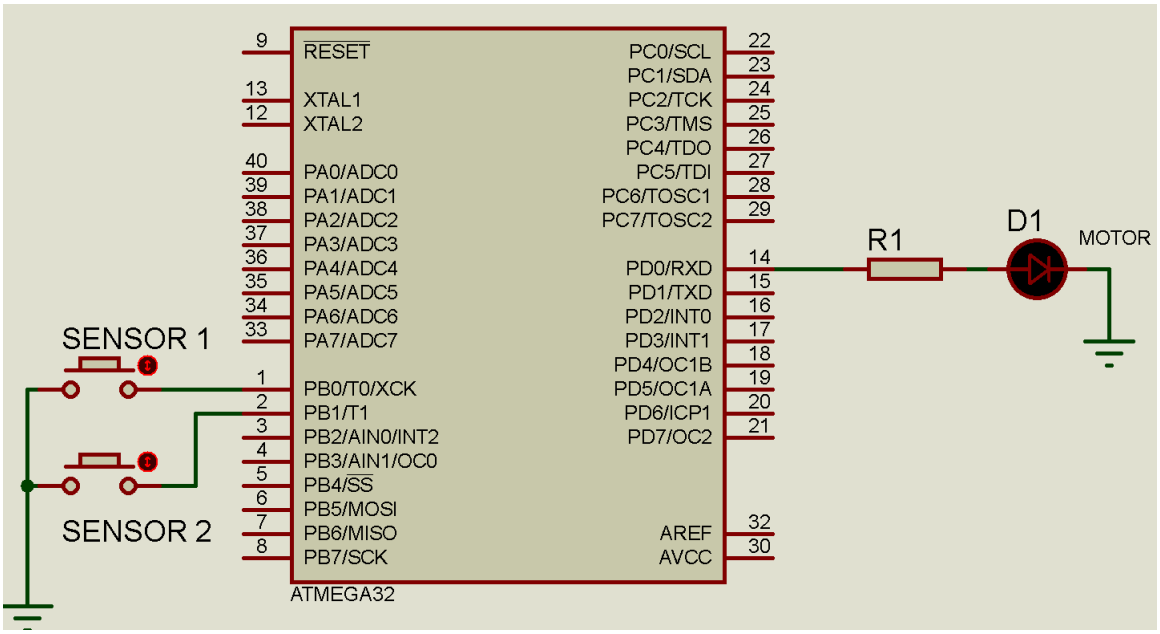

FIGURE 9.6: Road automatic bump schematic

Example 9.3.1

Write an AVR C program to get a byte of data from Port B, and then send it to Port C.

Solution:

```
#include <avr/io.h>          //standard AVR header
int main(void)
{
    unsigned char temp;
    DDRB = 0x00;             //Port B is input
    DDRC = 0xFF;             //Port C is output
    while(1)
    {
        temp = PINB;
        PORTC = temp;
    }
    return 0;
}
```

Example 9.3.2

Write an AVR C program to get a byte of data from Port C. If it is less than 100, send it to Port B; otherwise, send it to Port D.

Solution:

```
#include <avr/io.h>                //standard AVR header
int main(void)
{
    DDRC = 0;                      //Port C is input
    DDRB = 0xFF;                   //Port B is output
    DDRD = 0xFF;                   //Port D is output

    unsigned char temp;
    while(1)
    {
        temp = PINC;               //read from PINC
        if ( temp < 100 )
            PORTB = temp;
        else
            PORTD = temp;
    }
    return 0;
}
```

Homework

1. Write a program of Road automatic speed bump
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.4 Laboratory work 4

Locker with Keypad

Objectives of laboratory work

- Understanding the parallel port as input
- Developing practical skills
- Learn to program parallel ports
- Understanding the work of keypad and learn to program it

Laboratory assignments

Using ATmega32 connected with 4x4 Keypad, write a program of the door locks. To open the door you need to put right 4 digits in rights sequence. If you enter the right code Green lights will flash and if you enter the wrong code Red light will flash.

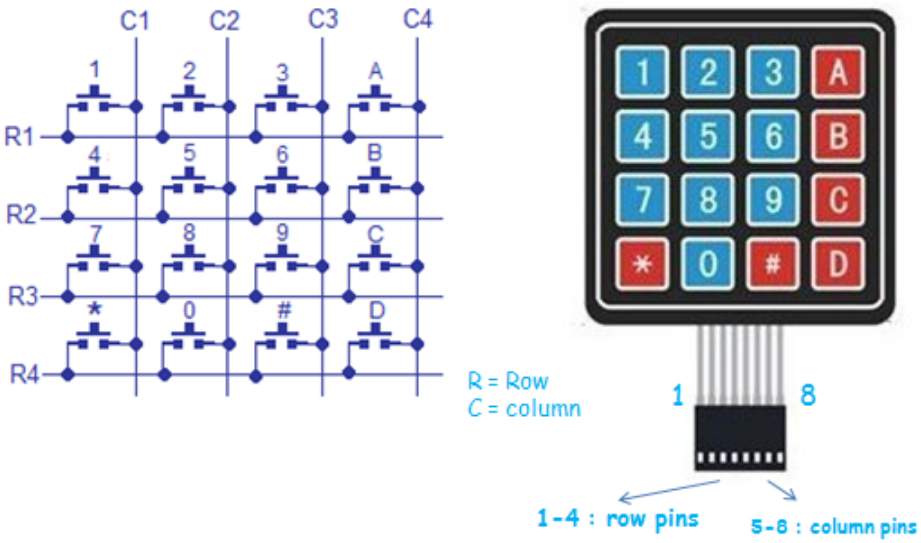

FIGURE 9.7: 4x4 Keypad schematic

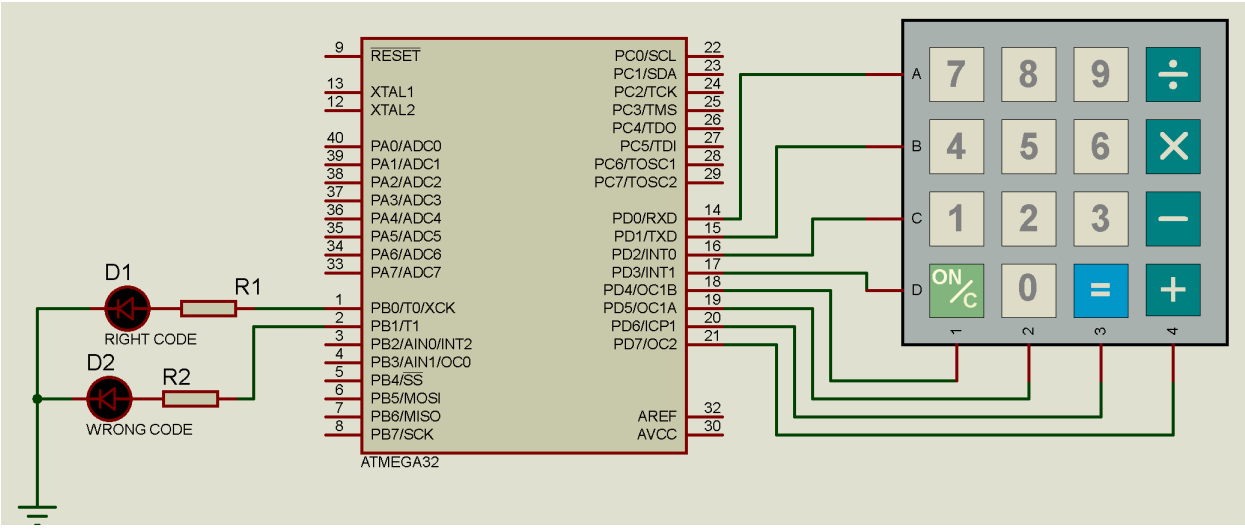

FIGURE 9.8: 4x4 Keypad schematic

Example 9.4.1

Write an AVR C program to toggle only bit 4 of Port B continuously without disturbing the rest of the pins of Port B.

Solution:

```
#include <avr/io.h>                    //standard AVR header
int main(void)
{
    DDRB = 0xFF;                       //PORTB is output
    while(1)
    {
        PORTB = PORTB | 0b00010000;    //set bit 4 (5th bit) of PORTB
        PORTB = PORTB & 0b11101111;    //clear bit 4 (5th bit) of PORTB
    }
    return 0;
}
```

Example 9.4.2

Write an AVR C program to monitor bit 5 of Port C. If it is HIGH, send 0x55 to Port B, otherwise, send 0xAA to Port B.

Solution:

```
#include <avr/io.h>                 //standard AVR header
int main (void)
{
    DDRB = 0xFF;                    //PORTB is output
    DDRC = 0x00;                    //PORTC is input
    DDRD = 0xFF;                    //PORTD is output
    while(1)
    {
    if (PINC & 0b00100000)          //check bit 5 (6th bit) of PINC
        PORTB = 0x55;
    else
        PORTB = 0xAA;
    }
    return 0;
}
```

Example 9.4.3

Write an AVR C program to read pins 1 and 0 of Port B and issue an ASCII character to Port D according to the following table:

pin1	pin0	
0	0	send '0' to Port D
0	1	send '1' to Port D
1	0	send '2' to Port D
1	1	send '3' to Port D

Solution:

```
#include <avr/io.h>                 //standard AVR header
int main (void)
{
    unsigned char z;
    DDRB = 0;                       //make Port B an input
    DDRD = 0xFF;                    //make Port D an output
    while(1)                        //repeat forever
    {
        z = PINB;                   //read PORTB
        z = z & 0b00000011;         //mask the unused bits
        switch(z)                   //make decision
        {
        case (0):
```

```
14          {
15              PORTD = '0';          //issue ASCII 0
16              break;
17          }
18          case (1):
19          {
20              PORTD = '1';          //issue ASCII 1
21              break;
22          }
23          case (2):
24          {
25              PORTD = '2';          //issue ASCII 2
26              break;
27          }
28          case (3):
29          {
30              PORTD ='3';           //issue ASCII 3
31              break;
32          }
33          }
34      }
35      return 0;
36  }
```

Homework

1. Write a program of Locker
2. Draw program algorithm of the program.

9.5 Laboratory work 5

DC Motor Control

Objectives of laboratory work

- Understanding the parallel port as output and input
- Developing practical skills
- Learn to program parallel ports
- Understanding the work of DC motor

Laboratory assignments

Using ATmega32 connected to a small DC motor and with two buttons write a program that changes the rotation direction of the motor. If button 1 pressed then the motor rotates in a clockwise direction and if button 2 pressed motor rotates counter-clockwise direction.

FIGURE 9.9: DC motor

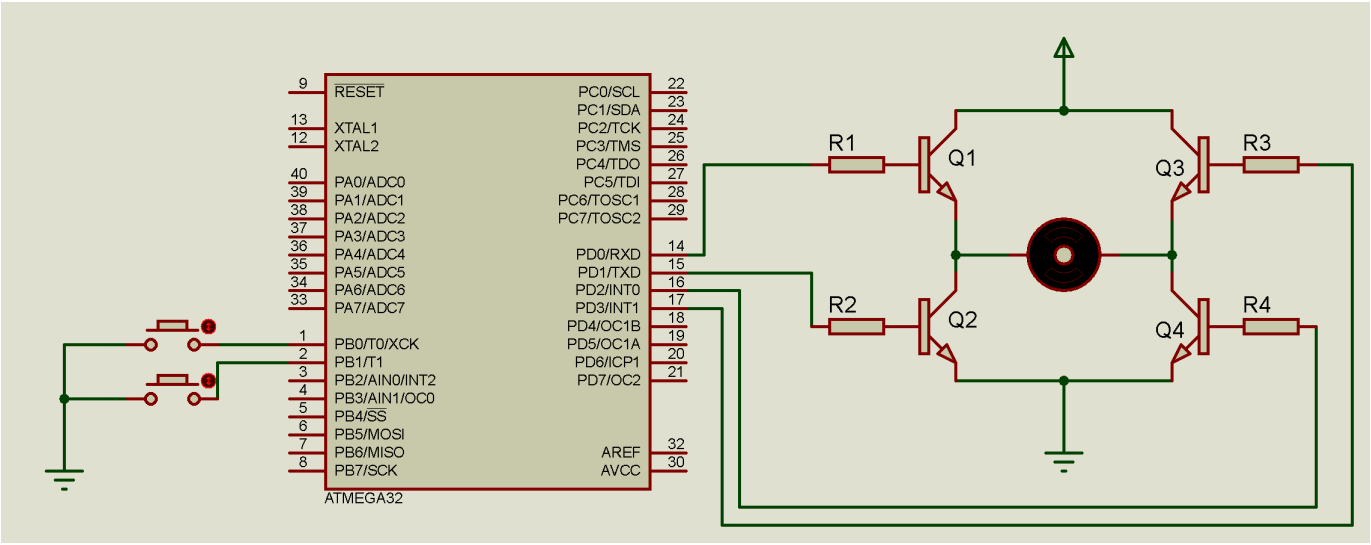

FIGURE 9.10: DC motor schematic

Example 9.5.1

The button is connected to bit 1 of Port B, and a motor is connected to bit 7 of Port C. Write an AVR C program,when button is pressed, turn on the motor.

Solution:

```
#include <avr/io.h>                        //standard AVR header
int main(void)
{
    DDRB = DDRB & 0b11111101;              //pin 1 of Port B is input
    DDRC = DDRC | 0b10000000;              //pin 7 of Port C is output
    while(1)
    {
    if (PINB & 0b00000010)                 //check pin 1 (2nd pin) of PINB
        PORTC = PORTC | 0b10000000;        //set pin 7 (8th pin) of PORTC
    else
        PORTC = PORTC & 0b01111111;        //clear pin 7 (8th pin) of PORTC
    }
    return 0;
}
```

Example 9.5.2

Solution:

```
#include <avr/io.h>                        //standard AVR header
#define MOTOR 7
#define BUTTON 1
int main(void)
{
    DDRB = DDRB & ~(1 << BUTTON);          //BUTTON pin is input
    DDRC = DDRC | (1 << MOTOR);            //MOTOR pin is output
    while(1)
    {
    if (PINB & (1<< BUTTON))               //check BUTTON pin of PINB
        PORTC = PORTC | (1 << MOTOR);      //set MOTOR pin of Port C
    else
        PORTC = PORTC & ~( 1<< MOTOR);     //clear MOTOR pin of Port C
    }
    return 0;
}
```

Homework

1. Write a program of DC motor control
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.6 Laboratory work 6

Stepper Motor Control

Objectives of laboratory work

- Understanding the parallel port as output
- Developing practical skills
- Learn to program parallel ports
- Understanding the work of Stepper motor

Laboratory assignments

Using ATmega32 connected to a small Stepper motor and with two buttons write a program that changes the rotation direction of the motor. If button 1 pressed then the motor rotates in a clockwise direction and if button 2 pressed motor rotates counter-clockwise direction. Write a program that Stepper motor rotates in Full step and Half step.

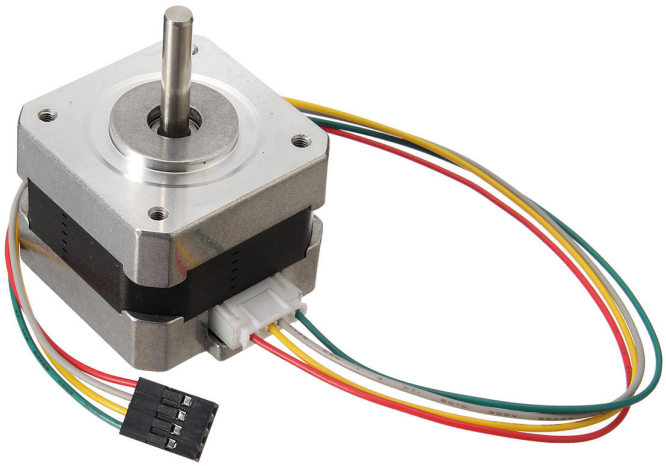

FIGURE 9.11: Stepper motor

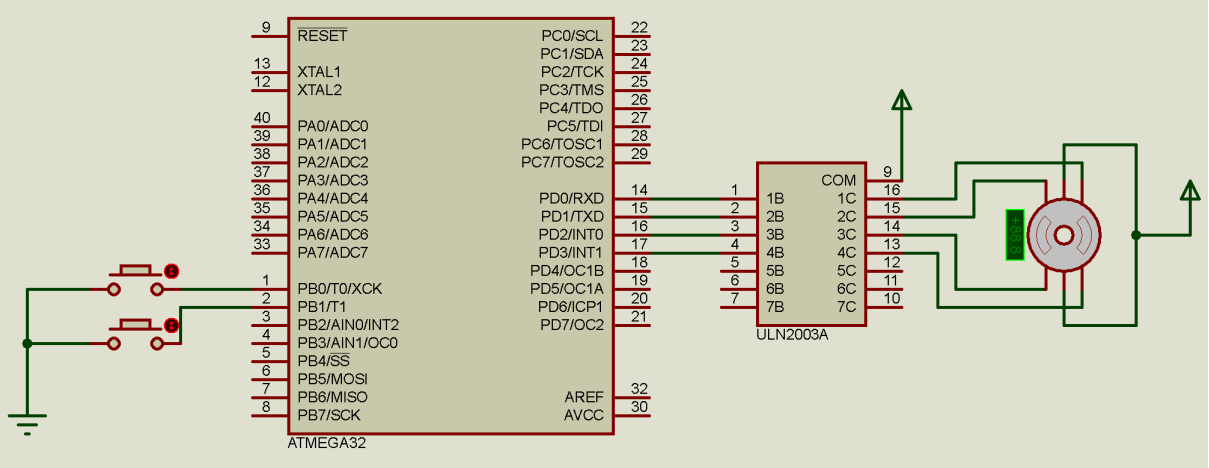

FIGURE 9.12: Stepper motor

Example 9.6.1

A switch is connected to pin PA7. Write a C program to monitor the status of SW and perform the following:

(a) If SW = 0, the stepper motor moves clockwise.

(b) If SW = 1, the stepper motor moves counterclockwise.

Solution:

```
#define F_CPU 8000000UL          //XTAL = 8 MHz
#include "avr/io.h"
#include "util/delay.h"
int main ()
{
    DDRA = 0x00;
    DDRB = 0xFF;
    while (1)
    {
        if ( (PINA & 0x80) == 0)
        {
            PORTB = 0x66;
            _delay_ms (100);
            PORTB = 0xCC;
            _delay_ms (100);
            PORTB = 0x99;
            _delay_ms (100);
            PORTB = 0x33;
            _delay_ms (100);
        }
        else
        {
            PORTB = 0x66;
```

```
24            _delay_ms (100);
25            PORTB = 0x33;
26            _delay_ms (100);
27            PORTB = 0x99;
28            _delay_ms (100);
29            PORTB = 0xCC;
30            _delay ms (100);
31        }
32    }
33 }
```

Homework

1. Write a program of Stepper motor control
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.7 Laboratory work 7

Seven Segments Display

Objectives of laboratory work

- Understanding the parallel port as output and input
- Developing practical skills
- Understanding dynamic lighting

Laboratory assignments

Using ATmega32 connected with 4 seven-segment LEDs, write a program that counts pressed key value. When SW1 pressed the value is incremented and displayed in the seven-segment display. When SW2 pressed the value is decremented and displayed in the seven-segment display.

Example 9.7.1

A seven-segment display is a form of an electronic display device for displaying decimal numbers. There are mainly two types of 7 segments display available. In a simple LED package, typically all of the cathodes (negative terminals) or all of the anodes (positive terminals) of the segment LEDs are connected and brought out to a common pin; this is referred to as a "common cathode" or "common anode" device.

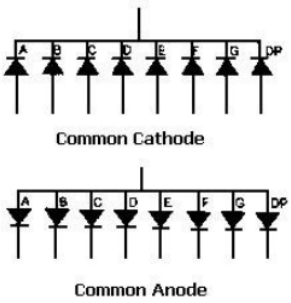

FIGURE 9.13: Types of seven segment

Seven segments each led has own letter.

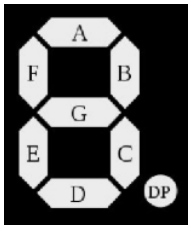

FIGURE 9.14: Seven segment

So we need program this 8 pin of PORTA to display our number on 7 segment display. Following table shows the hex code of displaying digits with common cathode led.

Digit	Hex code PORTA	A PA0	B PA1	C PA2	D PA3	E PA4	F PA5	G PA6	DT PA7
0	0x37	1	1	1	1	1	1	0	0
1	0x06	0	1	1	0	0	0	0	0
2	0x5B	1	1	0	1	1	0	1	0
3	0x4F	1	1	1	1	0	0	1	0
4	0x66	0	1	1	0	0	1	1	0
5	0x6D	1	0	1	1	0	1	1	0
6	0x7D	1	0	1	1	1	1	1	0
7	0x07	1	1	1	0	0	0	0	0
8	0x7F	1	1	1	1	1	1	1	0
9	0x6F	1	1	1	1	0	1	1	0

Connection with ATmega32 shown below.

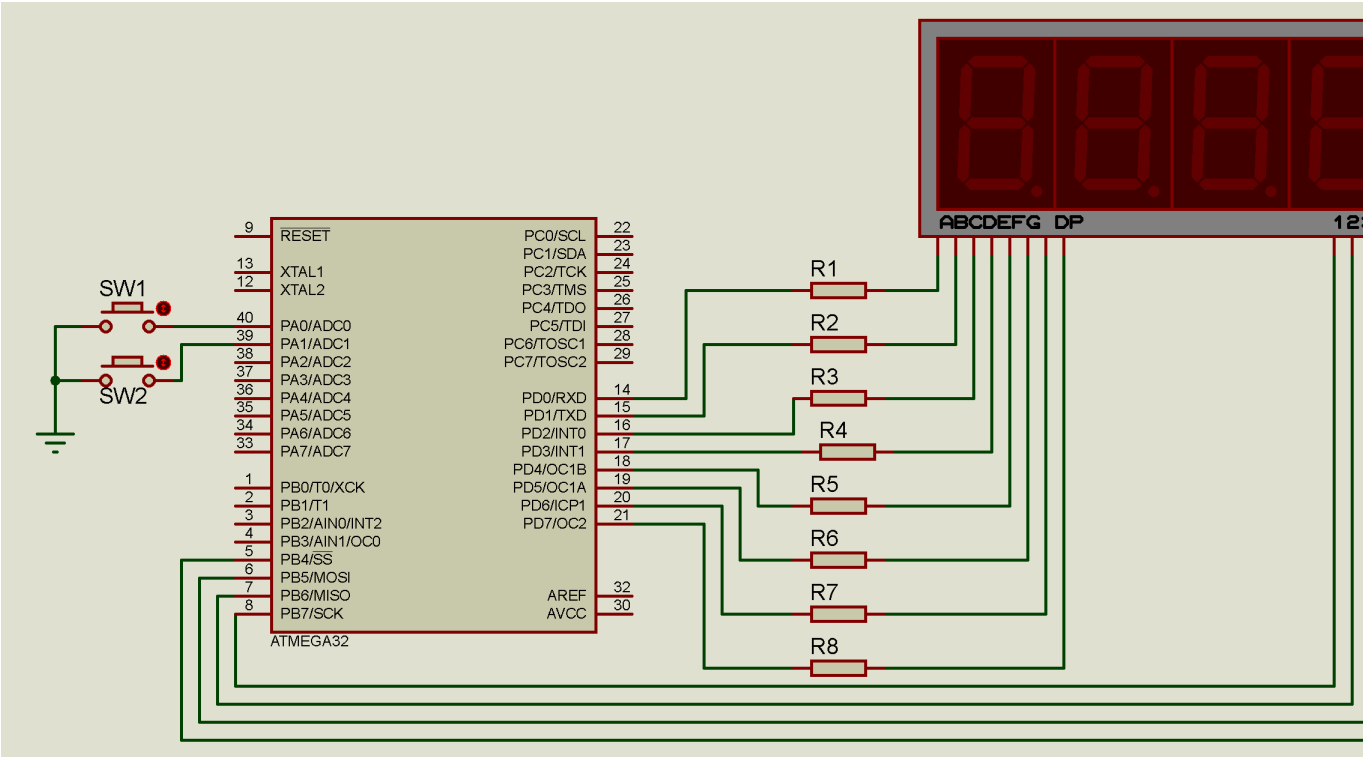

FIGURE 9.15: Seven segment connected with ATmega32

```
#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>
#define LED_Direction DDRA   // define LED Direction
#define LED_PORT PORTA       // define LED port

int main(void)
{
    LED_Direction |= 0xff;   // define LED port direction is output
    LED_PORT = 0xff;

    char array[]={0x37,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
                             // write hex value for CA display from 0 to
                                9
    while(1)
    {
        for(int i=0;i<10;i++)
        {
            LED_PORT = array[i]; // write data on to the LED port
            _delay_ms(1000);   // wait for 1 second
        }
    }
}
```

Homework

1. Write a program of Seven Segments Display
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.8 Laboratory work 8

LCD Control

Objectives of laboratory work

- Understanding the parallel port as output and input
- Developing practical skills
- Learn to program parallel ports
- Learn to program LCD display

Laboratory assignments

Using ATmega32 connected with LCD display, write a program that counts pressed key value. When SW1 pressed the value is incremented and displayed in the LCD display. When SW2 pressed the value is decremented and displayed in the LCD display.

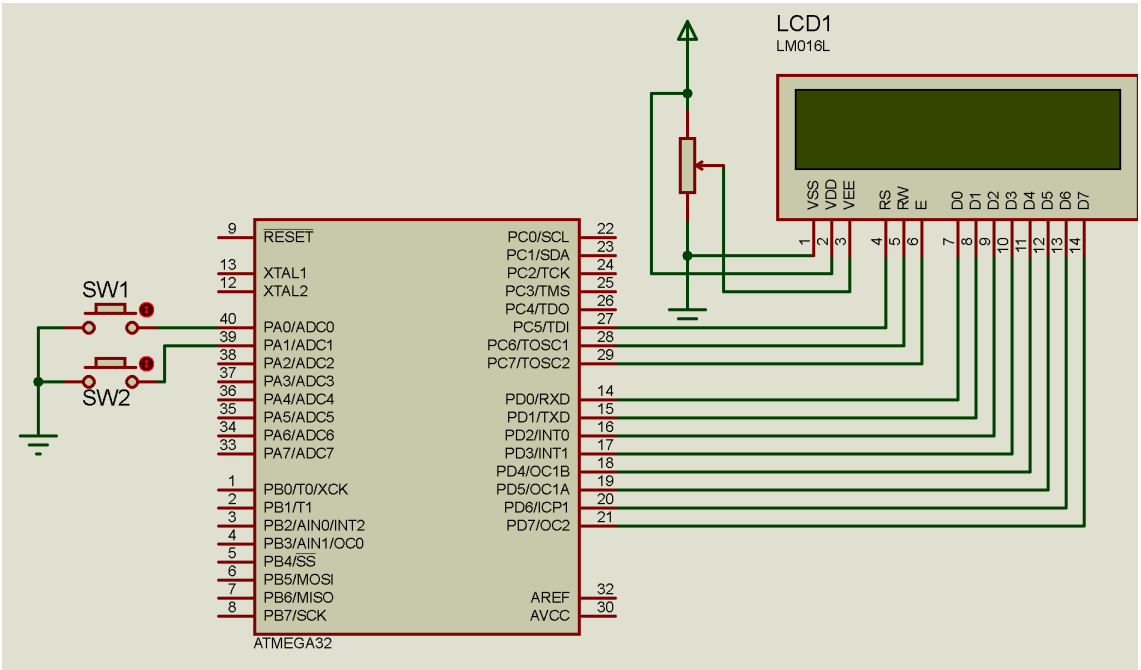

FIGURE 9.16: LCD display schematic

Example 9.8.1

```
#include <avr/io.h>                    //standard AVR header
#include <util/delay.h>                //delay header
#define LCD_DPRT PORTA                 //LCD DATA PORT
#define LCD_DDDR DDRA                  //LCD DATA DDR
#define LCD_DPIN PINA                  //LCD DATA PIN
#define LCD_CPRT PORTB                 //LCD COMMANDS PORT
#define LCD_CDDR DDRB                  //LCD COMMANDS DDR
#define LCD_CPIN PINB                  //LCD COMMANDS PIN
#define LCD_RS 0                       //LCD RS
#define LCD_RW 1                       //LCD RW
#define LCD_EN 2                       //LCD EN

void lcdCommand ( unsigned char cmnd )
{
LCD_DPRT = cmnd;                       //send cmnd to data port
LCD_CPRT &= ~ (1<<LCD_RS);             //RS = 0 for command
LCD_CPRT &= ~ (1<<LCD_RW);             //RW = 0 for write
LCD_CPRT |= (1<<LCD_EN);               //EN = 1 for H-to-L pulse
_delay_us(1);                          //wait to make enable wide
LCD_CPRT &= ~(1<<LCD_EN);              //EN = 0 for H-to-L pulse
_delay_us(100);                        //wait to make enable wide
}

 void lcdData ( unsigned char data )
{
LCD_DPRT = data;                       //send data to data port
LCD_CPRT |= (1<<LCD_RS);               //RS =1 for data
LCD_CPRT &= (1<<LCD_RW);               //RW = 0 for write
LCD_CPRT |= (1<<LCD_EN):               //EN =1 for H-to-L pulse
_delay us(1);                          //wait to make enable wide
LCD_CPRT & = ~(1<<LCD_EN);             //EN =0 for H-to-L pulse
_delay_us(100);                        //wait to make enable wide
}

void lcd_init()
{
LCD_DDDR = 0xFF;
LCD_CDDR = 0xFF;
LCD_CPRT &=~ (1<<LCD_EN);              //LCD EN = 0
_delay_us(2000);                       //wait for init.
lcdCommand(0x38);                      //init. LCD 2 line, 5 x 7 matrix
lcdCommand(0x0E);                      //display on, cursor on
lcdCommand(0x01);                      //clear LCD
_delay_us(2000);                       //wait
```

```
45 lcdCommand(0x06);                    //shift cursor right
46 }
47 void lcd_gotoxy(unsigned char x, unsigned char y)
48 {
49 unsigned char firstCharAdr[]={0x80,0xC0,0x94,0xD4};
50 lcdCommand(firstCharAdr[ y-1] + x - 1);
51 _delay_us(100);
52 }
53
54 void lcd_print( char * str )
55 {
56 unsigned char i = 0;
57     while(str[ i] !=0)
58 {
59 lcdData(str[i]);
60     i++ ;
61 }
62 }
63
64 int main (void)
65 {
66 lcd_init ();
67 lcd_gotoxy(1,1);
68 lcd_print("The world is but");
69 lcd_gotoxy(1,2);
70 lcd_print("one country");
71 while(1);                           //stay here forever
72 return 0;
73 }
```

Homework

1. Write a program of LCD Display
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.9 Laboratory work 9

Human Counting Device

Objectives of laboratory work

- Understanding the counter
- Learn to program counter
- Understanding registers of Timer and Counters

Laboratory assignments

The laser sensor is connected to a counter pin of the ATmega32 pin. If laser light is cut off then the pulse of that signal counted and displayed on the LCD display.
The second work is that DC motor has a rotary encoder and its output signal is connected to the counter pin of ATmega32. Show rotation count and rotation speed of the DC motor on the LCD display.

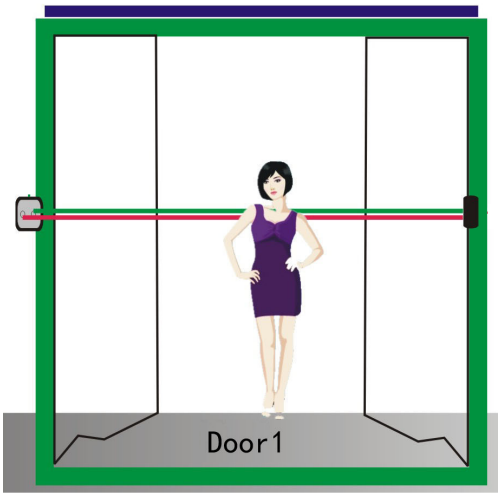

FIGURE 9.17: Human counter system

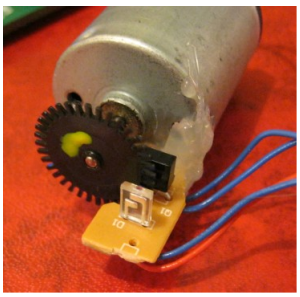

FIGURE 9.18: DC motor with Rotation Encoder

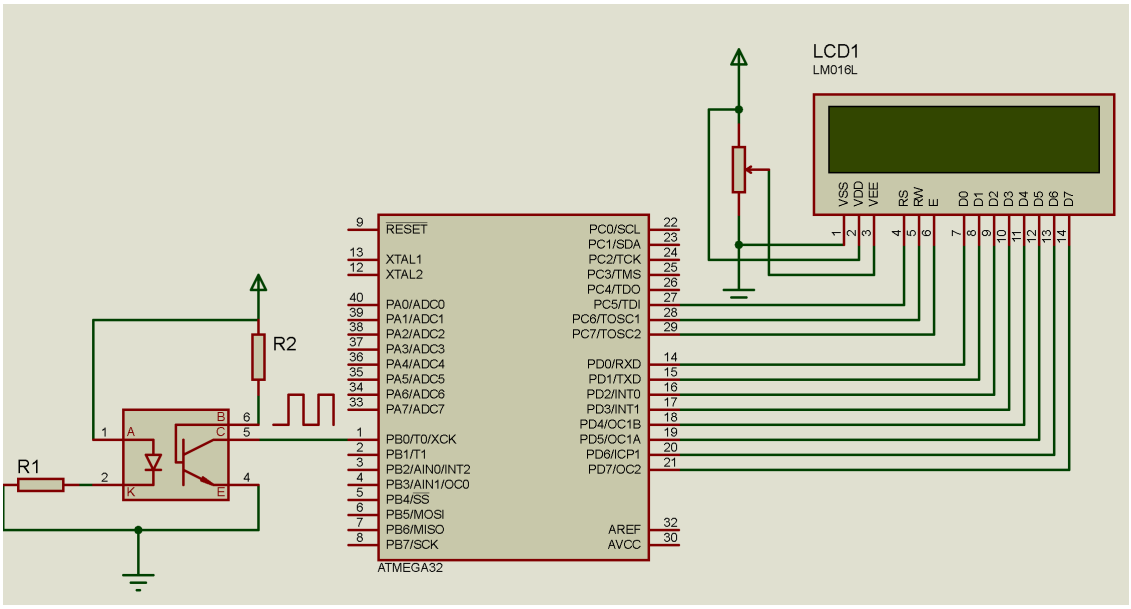

FIGURE 9.19: Human counter or rotation counter schematic

Example 9.9.1

Assuming that a 1 Hz clock pulse is fed into pin T0, use the TOV0 flag to extend Timer0 to a 16-bit counter and display the counter on PORTC and PORTD.

Solution:

```
#include "avr/io.h"
int main ( )
{
PORTB = 0x01;                    //activate pull-up of PB0
DDRC = 0xFF;                     //PORTC as output
DDRD = 0xFF;                     //PORTD as output
TCCR0 = 0x06;                    //output clock source
TCNT0 = 0x00;
while (1)
    {do
    {PORTC = TCNT0;}
    while((TIFR&(0x1<<TOV0))==0); //wait for TOV0 to roll over
    TIFR = 0xl<<TOV0;            //clear TOV0
    PORTD ++;                    //increment PORTD
    }
}
```

Homework

1. Write a program of Human counting device and draw program algorithm.

9.10 Laboratory work 10

Second meter

Objectives of laboratory work

- Understanding the timers of ATmega32
- Developing practical skills
- Learn to program timers
- Understanding registers of the timers

Laboratory assignments

Using ATmega32 connected with 3 buttons, write a program that counts seconds with millisecond precisions. When SW1 is pressed Start count, when SW2 pressed Stop count and when SW3 pressed Reset count. Use Timer0 and display count value on the LCD display.

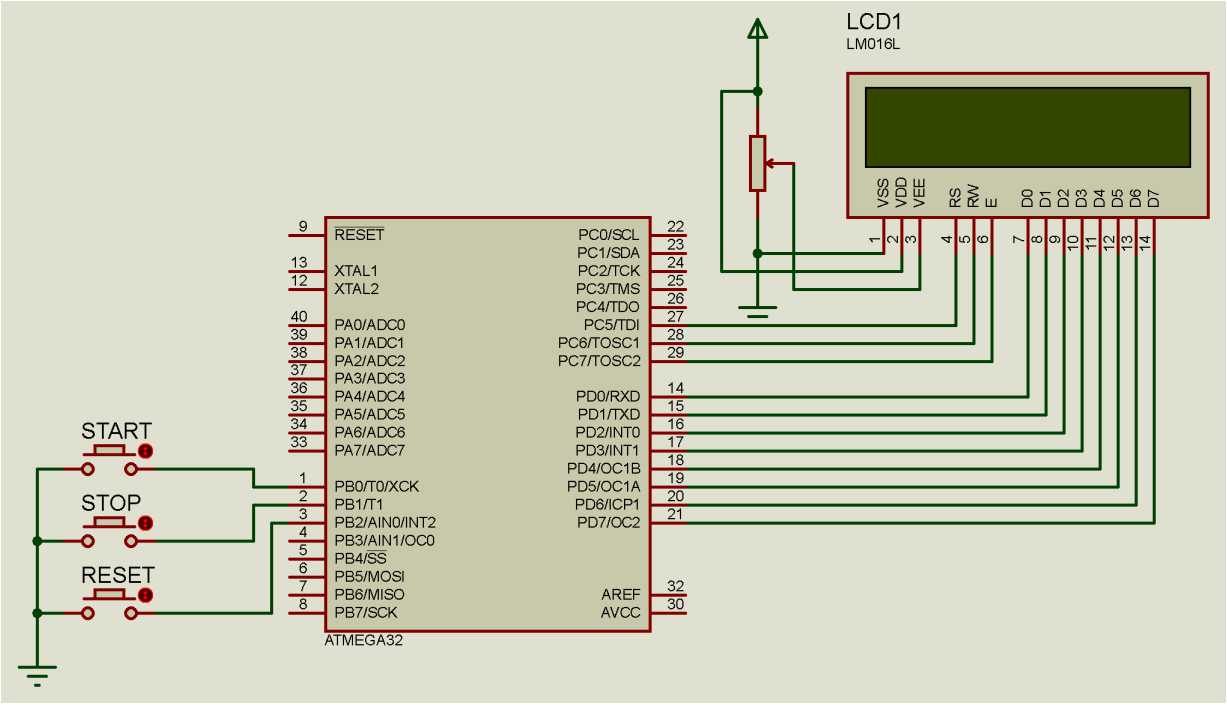

Figure 9.20: Second meter

Example 9.10.1

Write a C program to toggle all the bits of PORTB continuously with some delay. Use Timer0, Normal mode, and no prescaler options to generate the delay.

Solution:

```
#include "avr/io.h"
void T0Delay ( );
int main ( )
{
DDRB = 0xFF;                    //PORTB output port
    while (1)                   //repeat forever
    {
    PORTB = 0x55;
    T0Delay ( );                //delay size unknown
    PORTB = 0xAA;
    T0Delay ( );
    }
}
void T0Delay ( )
{
TCNT0 = 0x20;                   //load TCNT0
TCCR0 = 0x01;                   //Timer0, Normal mode, no prescaler
while ((TIFR&0x1)==0);          //wait for TF0 to roll over
TCCR0 = 0;
TIFR = 0xl;                     //clear TF0
}
```

Example 9.10.2

Write a C program to toggle only the PORTB.4 bit continuously every 70 us. Use Timer0, Normal mode, and 1:8 prescaler to create the delay. Assume XTAL = 8 MHz.

Solution:

Tmachine cycle = 1/8 MHz

Prescaler = 1:8

Tclock = 8 x 1/8 MHz = 1 us

70 us/1 us = 70 clocks

1 + 0xFF — 70 = 0x100 — 0x46 = 0xBA = 186

```
#include "avr/io.h"
void T0Delay ( );
int main ( )
{
    DDRB = 0xFF;                    //PORTB output port
    while (1)
    {
    T0Delay ( );                    //Timer0, Normal mode
```

```
9     PORTB = PORTB ^ 0x10;          //toggle PORTB.4
10    }
11 }
12
13 void T0Delay()
14 {
15 TCNT0= 186;                      //load TCNT0
16 TCCR0 = 0x02;                    //Timer0, Normal mode, 1:8 prescaler
17 while ((TIFR & (1<<TOV0))==0);   //wait for TOV0 to roll over
18 TCCR0 = 0;                       //turn off Timer0
19 TIFR = 0xl;                      //clear TOV0
20 }
```

Homework

1. Write a program of Secondmeter
2. Draw program algorithm of the program.

9.11 Laboratory work 11

Elevator Emergency System

Objectives of laboratory work

- Understanding the ATmega32 external interrupts
- Developing practical skills
- Learn to program external interrupts
- Understanding registers of external interrupts

Laboratory assignments

Using ATmega32 connected with 8 LEDs, write a program of the elevator. Imagine that each LED indicates the floor of the building, first led start blink and after 3 seconds second led start blink and it continues until reach to the eight led like elevator goes up and down. If the emergency button is pressed interrupt the main program and turn on emergency light.

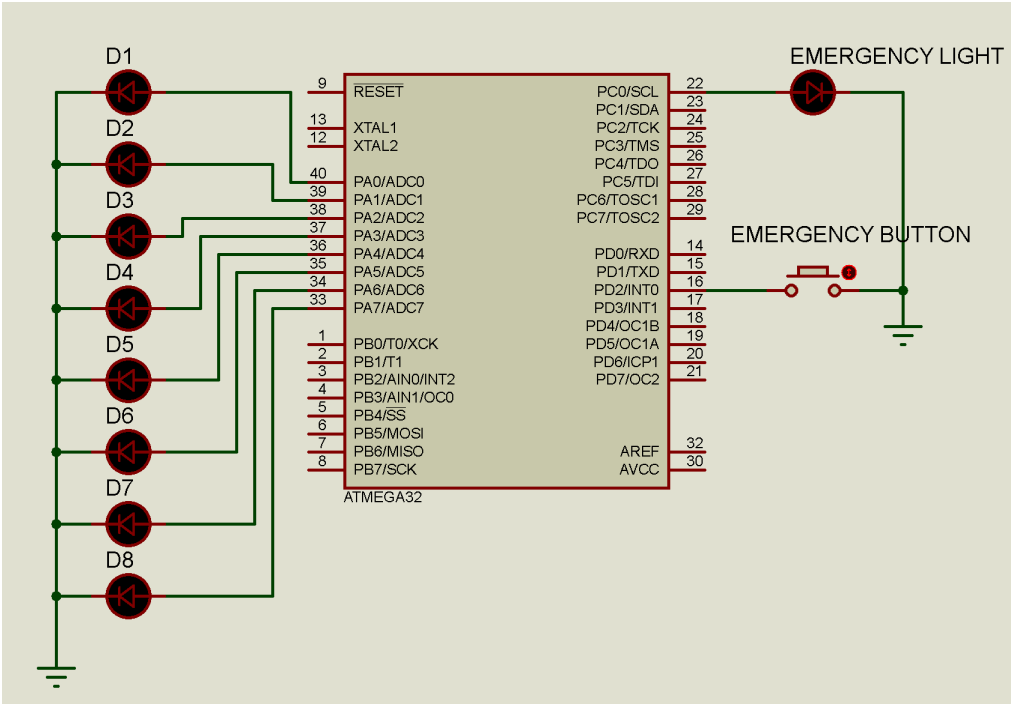

Figure 9.21: Elevator emergency system

Example 9.11.1

Assume that the INT0 pin is connected to a switch that is normally high. Write a program that toggles PORTC.3, whenever INT0 pin goes low. Use the external interrupt in level-triggered mode.

Solution:

```
#include "avr/io.h"
#include "avr/interrupt.h"
int main ()
{
    DDRC = 1<<3;              //PC3 as an output
    PORTD = 1<<2;             //pull-up activated
    GICR = (1<<INT0);         //enable external interrupt 0
    sei ();                   //enable interrupts
    while (1);                //wait here
}

ISR (INT0_vect)               //ISR for external interrupt 0
{
PORTC ^= (1<<3);              //toggle PORTC.3
}
```

Homework

1. Write a program of Elevator emergency system
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.12 Laboratory work 12

Temperature Controller

Objectives of laboratory work

- Understanding the ADC of ATmega32
- Developing practical skills
- Learn to program ADC of ATmega32
- Understanding registers of ADC

Laboratory assignments

Thermister connected to the ADC port of the ATmega32 controller. Read the analog value of the thermister using ADC and take a characteristic of the sensor. Imagine that this is a closed-loop system. If temperature greater than 30°C turn on the motor using P control. Display temperature value on the LCD display.

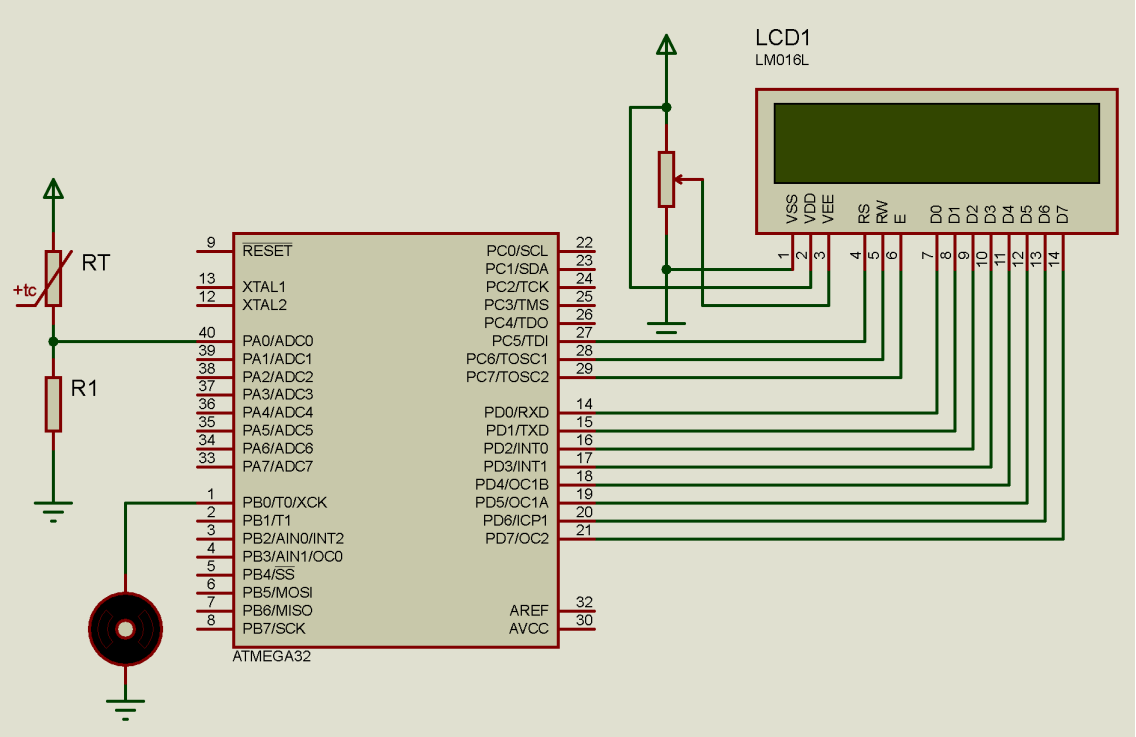

FIGURE 9.22: Temperature controller schematic

Example 9.12.1

```
#include<avr io.h>                          //standard AVR header
int main (void)
{
    DDRB = 0xFF;                           //make PORTB an output
    DDRD = 0xFF;                           //make PORTD an output
    DDRA = 0;                              //make PORTA an input for ADC
       input
    ADCSRA= 0x87;                          //make ADC enable and select ck
       /128
    ADMUX= 0xC0;                           //2.56V Vref, ADC0 single ended
       input
                                           //data will be right-justified
    while (1)
    {
        ADCSRA | = (1<<ADSC);              //start conversion
        while ((ADCSRA & (1<<ADIF)) == 0); //wait for conversion to finish
        PORTD = ADCL;                      //give the low byte to PORTD
        PORTB = ADCH;                      //give the high byte to PORTB
    }
    return 0;
}
```

Homework

1. Write a program of Temperature controller
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.13 Laboratory work 13

White Line Follower

Objectives of laboratory work

- Understanding the ADC of ATmega32
- Developing practical skills
- Learn to program ADC of ATmega32
- Understanding to use the PID controller
- Understanding to use the photoresistor

Laboratory assignments

Using ATmega32 design white line following robot. The robot is differential wheeled and has two photoresistors sensor which detects the white line. Write a program that follows the white line using PID control. Imagine that it is a closed-loop system.

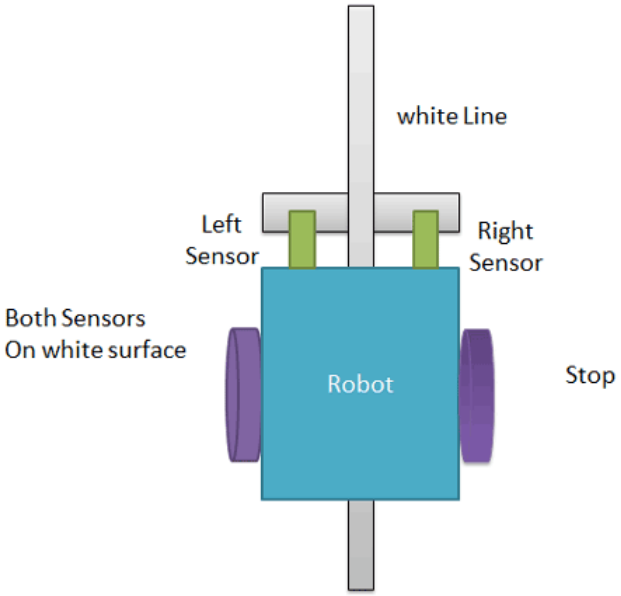

FIGURE 9.23: White line follower

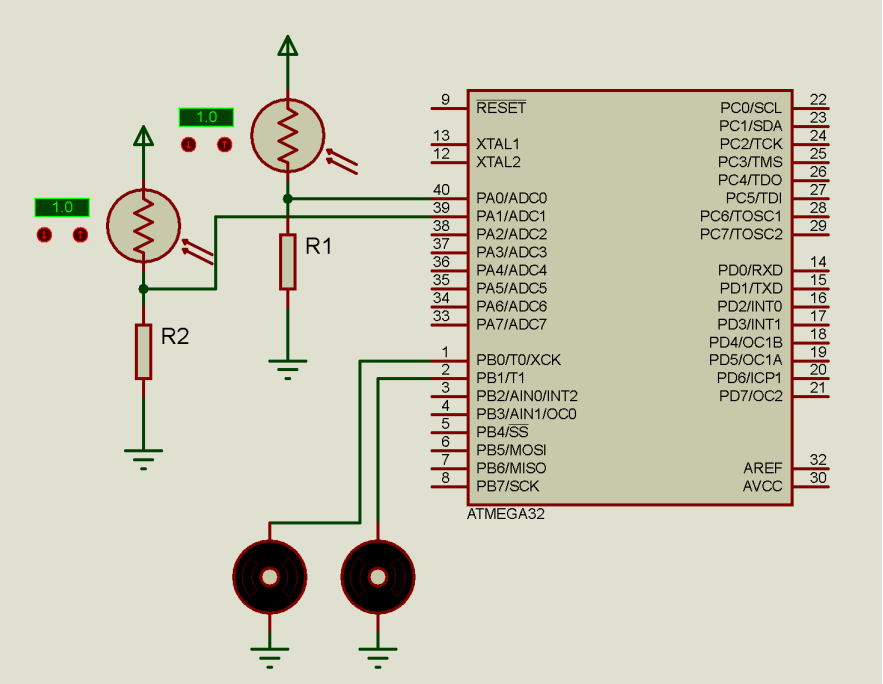

FIGURE 9.24: White line follower schematic

Example 9.13.1

With a couple light dependent resistors used as input, and a couple of motors driven by PWM as output. It's switching back and forth between the two photocells, and the reading from each one is assigned to the PWM compare value for its corresponding motor. This way a motor slows down when its photocell is covered.

```
#define F_CPU 1000000
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdint.h>

volatile const uint8_t adc2 = (1<<ADLAR) | 2;
volatile const uint8_t adc3 = (1<<ADLAR) | 3;

void initPWM ()
{
    DDRB |= (1 << PB0) | (1 << PB1);
    TCCR0A =
        (1 << COM0A1) |       // set OC0A on compare match, clear at TOP
        (1 << COM0B1) |       // set OC0B on compare match, clear at TOP
        (1 << WGM01) |        // fast PWM mode
        (1 << WGM00);
        TCCR0B = (1 << CS00); // prescaler = 1
}
```

```
20
21 void initADC ()
22 {
23     ADMUX = (1 << ADLAR) | (1 << MUX1);
24    ADCSRA =
25         (1 << ADEN) |         // Enable ADC
26         (1 << ADATE) |        // auto trigger enable
27         (1 << ADIE) |         // enable ADC interrupt
28         (1 << ADPS0) |        // Prescaler = 8
29         (1 << ADPS1);         //    - 125KHz with 1MHz clock
30         ADCSRB = 0;           // free running mode
31         sei();
32         ADCSRA |= (1 << ADSC);  // start conversions
33 }
34
35 ISR(ADC_vect)
36 {
37     static uint8_t firstTime = 1;
38     static uint8_t val;
39     val = ADCH;
40     if (firstTime == 1)
41         firstTime = 0;
42
43     else if (ADMUX == adc2)
44     {
45         ADMUX = adc3;
46         OCR0A = val;
47     }
48
49     else if (ADMUX == adc3)
50     {
51         ADMUX = adc2;
52         OCR0B = val;
53     }
54 }
55
56 int main ()
57 {
58     initPWM ();
59     initADC ();
60
61     for (;;)
62     {
63     }
64 }
```

Homework

1. Write a program of White line follower
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.14 Laboratory work 14

Stabilization of Rotational Speed of DC Motor

Objectives of laboratory work

- Understanding the PWM signal
- Developing practical skills
- Learn to program PID control
- Understanding registers of timer and counters

Laboratory assignments

Imagine that it is a closed-loop system. Using PID control do stabilization of the rotational speed of the DC motor with the PWM signal.

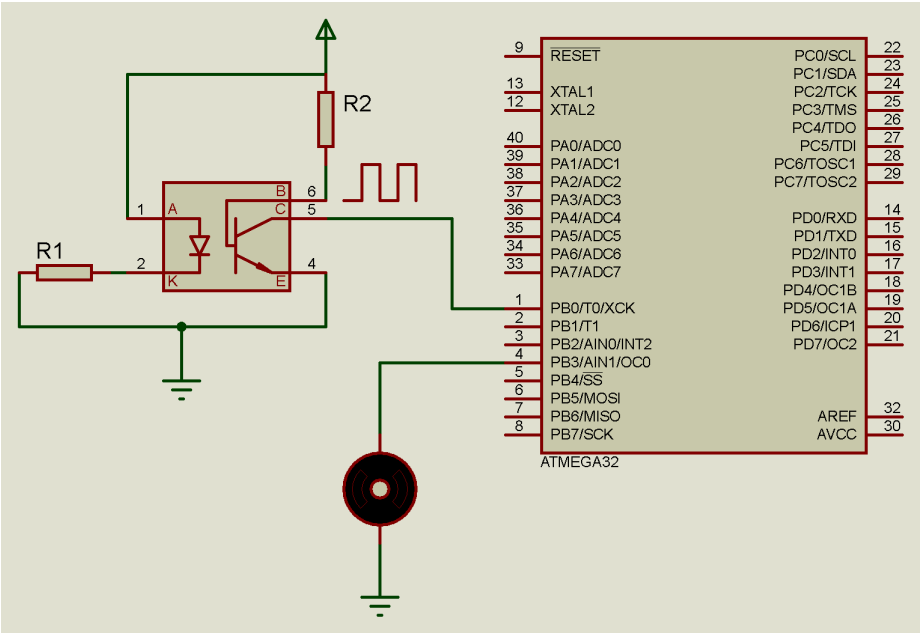

Figure 9.25: PID control for DC motor speed stabilization

Example 9.14.1

Solution:

```
#include "avr/io.h"
int main ()
{
    DDRB |= (1 << 3);
    OCR0 = 191;
    TCCR0 = 0x69;          //Fast PWM, no prescaler, non-inverted
    while (1);
    return 0;
}
```

Homework

1. Write a program of DC motor speed control
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.15 Laboratory work 15

Pendulum Stabilization

Objectives of laboratory work

- Understanding the PWM signal
- Developing practical skills
- Learn to program PID control
- Understanding registers of timer and counters
- Developing of the potentiometer as an angular sensor

Laboratory assignments

Using a potentiometer as an angular sensor do the stabilization of pendulum at a given angle. DC motor with propeller takes vertical take-off that changes the angle of pendulum.

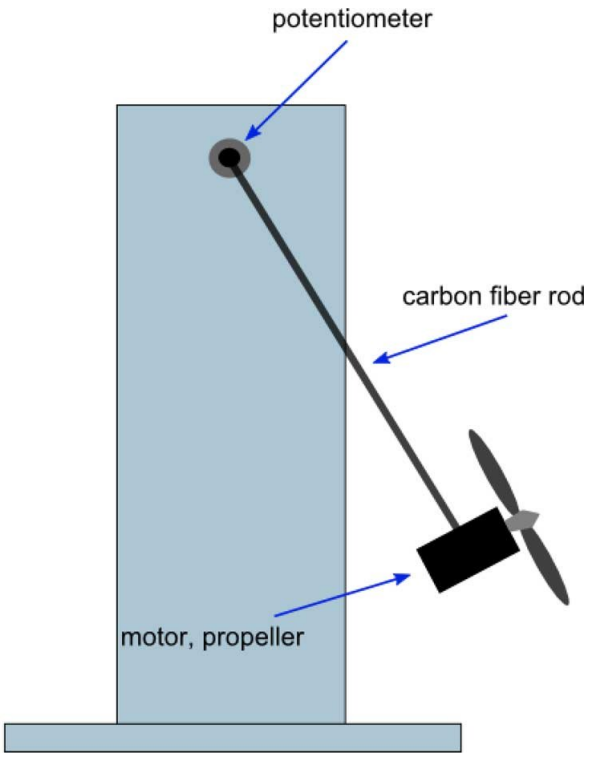

FIGURE 9.26: PID control for Pendulum

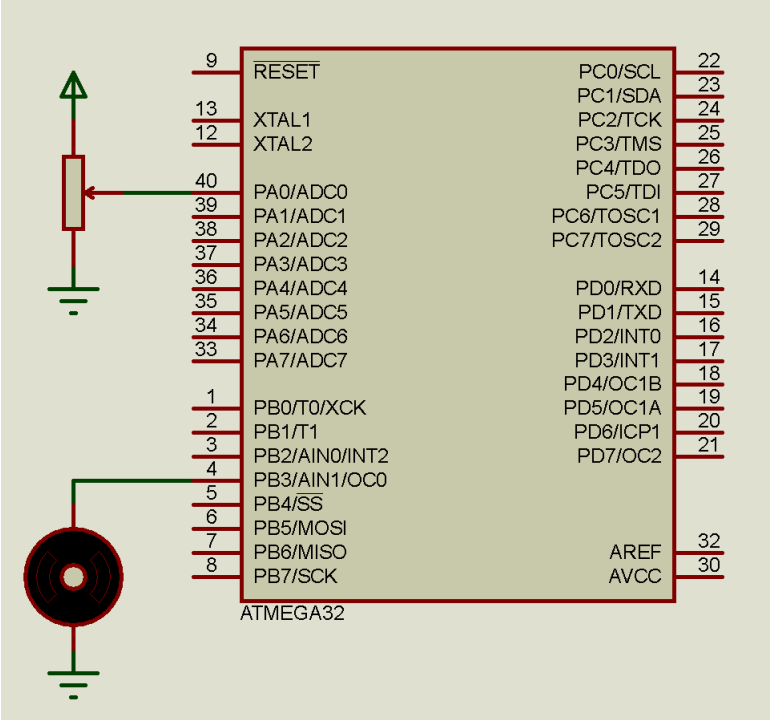

FIGURE 9.27: PID control for Pendulum schematic

Example 9.15.1

PID Algorithm generates a control variable from the current value, and the required value. Since the aim is to keep the motor speed constant. The first function reads (Read()) encoder value that measures the rotation speed of the motor. Second function writes (Write()) PWM value to control motor.

```
#include <avr/io.h>
int Kp = 2                                          //PID constants
int Ki = 5
int Kd = 1
unsigned long currentTime, previousTime;
double elapsedTime;
double Error;
double lastError;
double input, output, setPoint;
double cumError, rateError;
setPoint = 100;                                     //set point at zero
   degrees
int main ()
{
    while(1)
    {
        input = Read(A0);                           //read from rotary
           encoder connected to A0
        currentTime = millis();                     //get current time
```

```
18        elapsedTime = currentTime - previousTime;  //compute time (dT)
19        error = Setpoint - inp;                    // determine error
20        cumError += error * elapsedTime;           // compute integral
21        rateError = (error - lastError)/elapsedTime;  // compute
              derivative
22        output = Kp*error + Ki*cumError + Kd*rateError; //PID output
23        lastError = error;                         //remember current
              error
24        previousTime = currentTime;                //remember current time
25        Write(3, output);                          //control the motor
              based on PID value
26    }
27    return 0;
28 }
```

Homework

1. Write a program of Pendulum stabilization
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

9.16 Laboratory work 16

Serial Communication

Objectives of laboratory work

- Understanding the serial port
- Developing practical skills
- Learn to program serial port
- Understanding registers of USART port

Laboratory assignments

Connect two ATmega32 controllers using the USART port. Do asynchronous data transfer between those controllers.

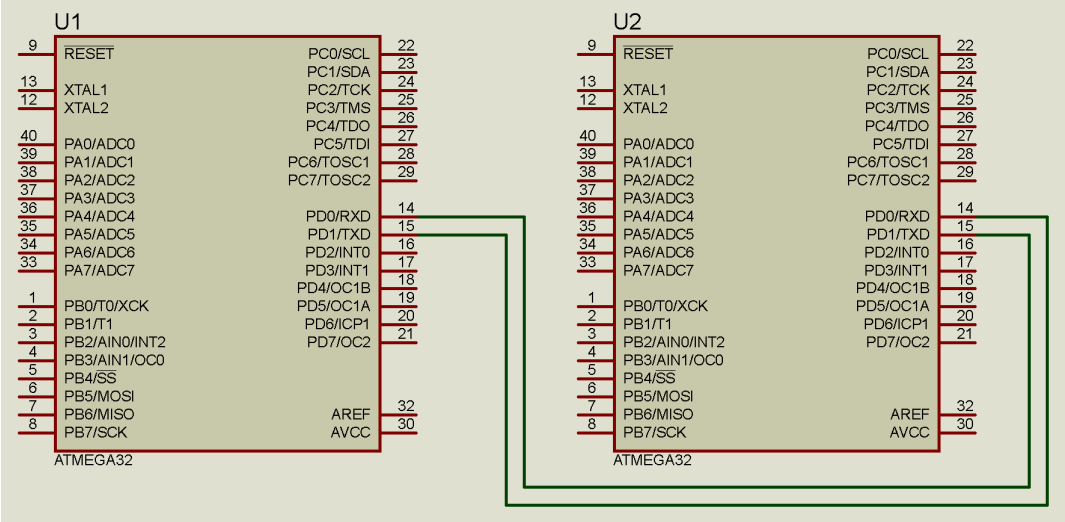

FIGURE 9.28: USART connection

Example 9.16.1

Write a C program for the AVR to transfer the letter 'G' serially at 9600 baud, continuously. Use 8-bit data and 1 stop bit. Assume XTAL = 8 MHz. Solution:

```
#include <avr/io.h>                          //standard AVR header
void usart_init (void)
{
    UCSRB = (1<<TXEN);
    UCSRC = (1<< UCSZ1) | (1<<UCSZ0) | (1<<URSEL);
```

```
    UBRRL = 0x33;
}
void usart_send (unsigned char ch)
{
    while (! (UCSRA & (1<<UDRE)));          //wait until UDR is empty
    UDR = ch;                               //transmit 'G'

}

int main (void)
{
    usart_init();                           //initialize the USART
    while (1)                               //do forever
        usart_send ('G');                   //transmit 'G' letter
    return 0;
}
```

Example 9.16.2

Program the AVR in C to receive bytes of data serially and put them on Port A. Set the baud rate at 9600, 8-bit data, and 1 stop bit.

Solution:

```
#include <avr/io.b>                         //standard AVR header
int main (void)
{
    DDRA = 0xFF;                            //Port A is input
    UCSRB = (1<<RXEN);                      //initialize USART
    UCSRC = (1<< UCSZ1) | (1<<UCSZ0) | (1<<URSEL);
    UBRRL = 0x33;
    while (1)
    {
        while (! (UCSRA & (1<<RXC)));       //wait until new data
        PORTA = UDR;
    }
 return 0;
}
```

Homework

1. Write a program of Serial communication
2. Draw program algorithm of the program.
3. Make this device as a closed-loop system and draw its a block diagram

Bibliography

[1] Ying Bai and Zvi S. Roth. "Classical and Modern Controls with Microcontrollers". In: *Springer* (2019).

[2] Norman S. Nise. "Control Systems Engineering". In: (2013).

[3] Nadim Maluf and Kirt Williams. "An Introduction to Microelectromechanical Systems Engineering". In: *Artech House, Inc* (2004).

[4] Vijay K. Varadan Julian W. Gardner and Osama O. Awadelkarim. "Microsensors MEMS and Smart Devices". In: *JOHN WILEY SONS, LTD* (2002).

[5] Sepehr Naimi Muhammad Ali Mazidi Sarmad Naimi. "The AVR microcontroller and embedded system using assembly and c". In: *Pearson* (2009).

[6] Yui May L. Chang and James Y. Shih. "Microprocessor Applications and Building Control Systems to Achieve Energy Conservation". In: *NATIONAL BUREAU OF STANDARDS* (1980).

[7] Joe Pardue. "C programming for Microcontrollers". In: *Published by Smiley Micros* (2005).

[8] Koovappady P O. "Lab manual – Embedded Systems AVR". In: ().

[9] Aravind E Vijayan. "A beginners Guide to AVR". In: *Technical Report* (2014).

[10] "Programming with AVR Microcontroller". In: *Research Design Lab* (2014).

[11] Gokaraju Rangaraju. "Microcontrollers Laboratory , work book". In: *Gokaraju Rangaraju Institute of Engineering and Technology* (1997).

[12] Panayotis Papazoglou. "An Educational Guide to the AVR Microcontroller Programming". In: *Kessariani, Athens, Greece* (2018).

[13] Shinsuke Hara. "Digital System Design — Use of Microcontroller". In: *River Publishess* (2010).