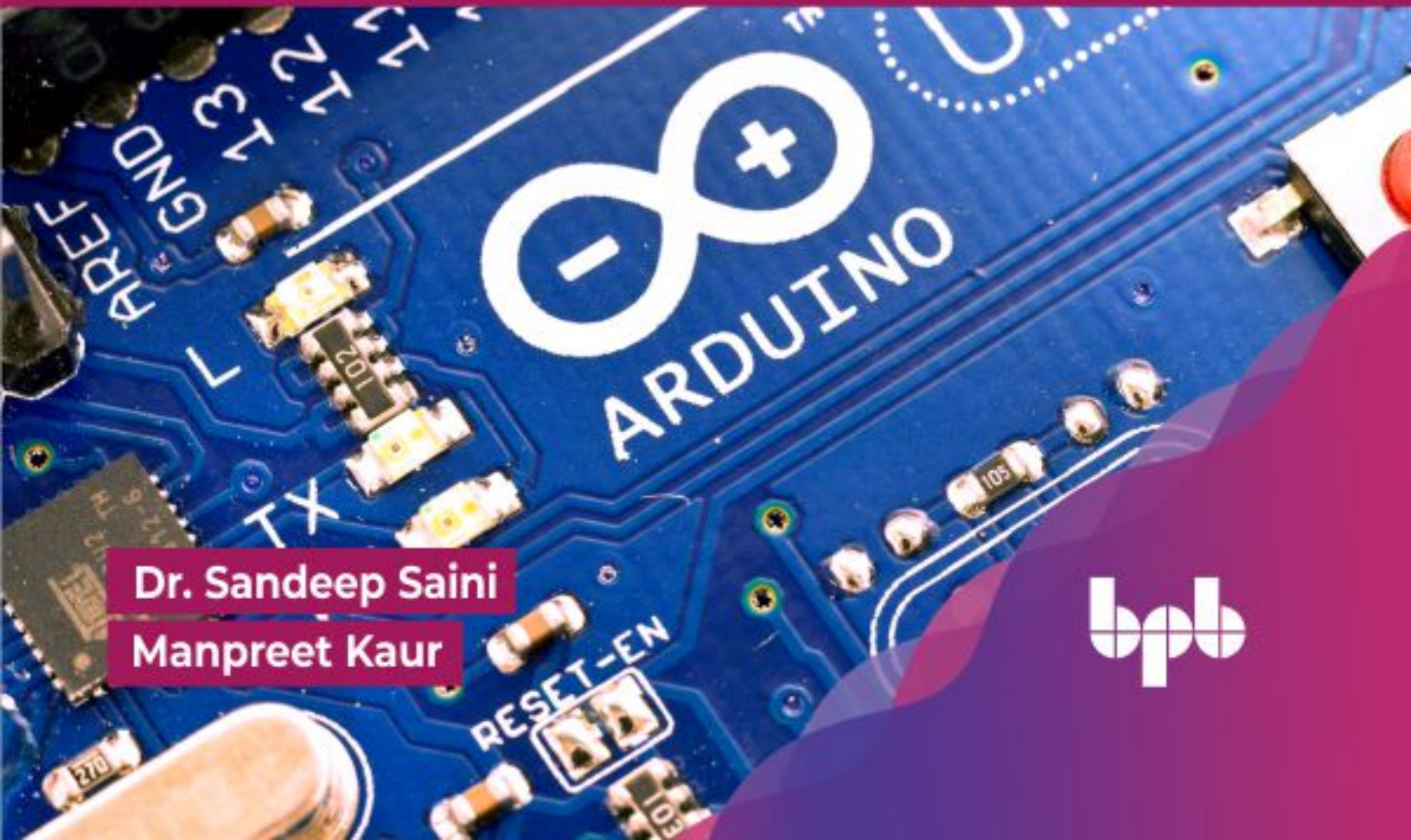
Arduino Solutions Handbook

Design interesting DIY projects using Arduino Uno, C and C++



Dr. Sandeep Saini
Manpreet Kaur

bpb

Arduino Solutions Handbook

Design interesting DIY projects using Arduino Uno, C and C++

Dr. Sandeep Saini
Manpreet Kaur

bpb

Arduino Solutions Handbook

Design Interesting DIY Projects Using Arduino Uno, C and C++

Dr. Sandeep Saini

Manpreet Kaur

www.bpbonline.com

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2023

Published by BPB Online

WeWork, 119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-5551-325-0

www.bpbonline.com

Dedicated to

Our beloved daughter

Dhriti Saini

About the Authors

Dr. Sandeep Saini is working as an Assistant Professor in the Department of Electronics and Communication Engineering at the LNMIIT Jaipur, India. He has taught Robotics and Electronics subjects at the university level in India and abroad for more than a decade. He has taught more than 8000 students in online and offline modes during these years.

He received his B.Tech. degree in Electronics and Communication Engineering from the International Institute of Information Technology (IIIT), Hyderabad, India, in 2008. He completed his MS from the same institute in 2010. He received his Ph.D. from Malaviya National Institute of Technology (MNIT), Jaipur, India.

Sandeep started the Robotics club at IIIT Hyderabad in 2006 and coordinated the Robotics and Phoenix club at the LNMIIT Jaipur. He has written 6 books and published 35 peer-reviewed journals and conference papers.

Manpreet Kaur is an ICT and Robotics enthusiast who loves to work with school kids at different levels. Manpreet has a post-graduate degree in Computer Science, specialising in Image Processing. She is associated with a top school in Jaipur as a Robotics teacher. She teaches Arduino, Scratch, Micro bit, and Python-based Programming and Game development concepts. She has taught in India and abroad for more than 4 years in international schools.

About the Reviewer

Balaji is a Microsoft Azure IoT specialty certified Embedded and IoT Developer. He has a significant amount of experience in the end-to-end processes of both the hardware and software sections of IoT. His area of expertise is in consumer electronics and automation. Quite recently, he expanded his horizons by stepping into Industry 4.0. He is currently exploring PLC, SCADA, and cybersecurity for OT in his free time. He is a holder of multiple Microsoft Azure specific certifications.

He specializes in wireless sensor networks and enjoys working with new development boards and technologies. His areas of interest are not limited to low-power devices, data analytics, and ML on edge.

He is currently working on Matter supported devices and Alexa based IoT devices.

Acknowledgement

This book is based on Arduino-based project development. We teach this topic at our respective institutes, and this book was an excellent opportunity to compile all the ideas into a well organised book. Both of us teach robotics and related subjects at our respective institutes, and we have compiled our best projects in this book. First and foremost, I would like to thank my parents (L.C. Saini, Sudesh Saini, Sukhwinder Singh and Gurmeet Kaur) for continuously encouraging us to write the book. We could have never completed this book without their support.

My gratitude also goes to the team at BPB Publications for being supportive enough to provide me quite a long time to finish the first part of the book and also allow me to publish the book in multiple parts, since image processing, being a vast and very active area of research, it was impossible to deep-dive into a different class of problems in a single book, especially by not making it too voluminous. We would like to thank Balaji for his technical reviews and for helping us in correcting a few mistakes with his feedback.

Preface

Arduino is the most popular microcontroller across the globe. It is also one of the most flexible and feature-rich boards. A variety of hobby ad industrial applications are designed using Arduino. This book will contain a complete solution guide to design such applications comprehensively using the Arduino Uno board.

In this book, the learners will get complete information about doing a project using the Arduino board. The learning will start by knowing the different features of the board. You will learn about the programming environments and techniques using different operating systems. Sensor integration and data acquisition using different modes of communication will also be taught through detailed examples.

By the end of this book, the reader will become a sound Arduino developer with hands-on skills in using Arduino, various peripherals, electrical devices, tools, IDE, and some essential hold on C++ coding. The project implementation will give them the confidence to translate their own ideas into working prototypes using an Arduino board. The book will help the students design innovative solutions for their projects, and industry professionals will know about the board's advanced features.

This book is compiled into **4 different sections** with **10 chapters**. The first section is Programming and Arduino. This section introduces the learners to the Arduino board, its features, capabilities, usages, and the respective programming tools and techniques. The second section is dedicated to the sensors, displays and actuators. The user will learn to connect various sensors, actuators and displays to an Arduino board, read inputs, process them and control the output devices. The programming taught in the previous section will be extensively used to make useful recipes in this section. The third section teaches communication with other boards. We'll learn methods and code to connect with other microcontrollers through I2C, SPI and CAN bus communication protocols. The last section will teach you about wireless connectivity options using Arduino boards. We'll learn the connections using Bluetooth and Wi-Fi techniques.

Every chapter is divided into multiple recipes. These recipes are like individual projects and can be developed independently. But these recipes are arranged in such a manner that the knowledge of the user will build vertically with each recipe in that domain. The user will realize how the knowledge gained in the previous section; chapter of the recipe is being utilized in future recipes. The brief summary of all the chapters of the book is as follows:

Chapter 1 is the introductory chapter of the book. We have discussed the need for Arduino and its usefulness for prototyping projects. We have explained the process of installing the required software platforms for different operating systems. Th process of installing the Integrated Development Environment (IDE) on various operating systems is explained in detail. In the end, we have taught the installation of additional libraries to the IDE.

Chapter 2 introduces the learners to the basic coding concepts of Arduino Uno. Arduino can be programmed with different programming languages, but the default language is C and C++. The learners will learn the most basic commands and syntax of writing an Arduino program (which is called a sketch). The chapter will also explain the process of compiling and uploading the code to test on an Arduino board. The user will be able to control multiple LEDs through these recipes.

Chapter 3 focuses on advanced coding concepts with relevant recipes. This chapter will talk about conditional statements, loops, interrupts, PWM, Analog values, and tone generation. The recipes are arranged to enhance the user's knowledge of all possible aspects of coding an Arduino board.

Chapter 4 is the starting chapter of the next section and focuses on learning different display devices. Displaying the output of your system comprehensively is a vital component of the project. This chapter will introduce different display devices like LEDs, LCDs, 7-segment displays, Neo Pixel and OLED screens. We will explain their working principles and the connections to an Arduino board.

Chapter 5 focuses on sensor integration with an Arduino board. Arduino is primarily used to acquire data from various sources and then process it to generate an output. Sensors are the primary source for different types of

data collection. This chapter will introduce the working principles and the connections of different analog and digital sensors. These sensors acquire different environmental parameters like temperature, humidity, and moisture. These can also be used to measure distance, detect objects, and detect the intensity of light or the movement of a person. We will also introduce a keypad to provide inputs manually to the Arduino board.

Chapter 6 talks about motion. Motion is an integral part of a lot of Arduino-based projects. The field of robotics is highly dependent on different moving parts. This chapter will introduce the learners to DC and servo motors. They will learn the techniques to connect them and the additional tools required to control their movements.

Chapter 7 introduces the learner to the next section. This chapter will introduce the learners to different communication options with other boards and devices. In this section, we focus on I2C and SPI communication protocols. We can connect one Arduino board with another Arduino board or any other device that supports I2C communication. This communication uses fewer numbers of wires. SPI is a serial communication protocol. In this communication, devices act like masters and slaves. That means one device can control the other devices. In this chapter, we will learn the concepts of SPI communication and how to connect two Arduino boards in a master-slave configuration using SPI communication.

Chapter 8 talks about one of the most popular communication techniques used in modern vehicles, i.e. controller area network (CAN). CAN allows different devices connected to microcontrollers to communicate with each other inside a vehicle. We will learn how to use the CAN shield, MCP2515, to connect with an Arduino board and write sketches for the same.

Chapter 9 introduces us to wireless connectivity techniques. We will learn to connect and communicate using an HC-06 Bluetooth module with an Arduino board. We will learn how to connect a smartphone and Arduino through the same network of Bluetooth devices. A good number of exciting recipes are designed using the same concept.

Chapter 10 is the book's last chapter, and we talk about data transmission using the most widely used communication protocol, i.e. Wi-Fi. Arduino is not having any built-in option to connect via Wi-Fi. So, we will learn about

a new series of boards. We will learn about ESP32 and ESP8266. We will learn how to read internet data via Wi-Fi protocol using these boards. We will also learn how to control certain devices via Wi-Fi. This chapter also lays down the foundation of the Internet of Things (IoT) domain.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/8v9f3c0

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Arduino-Solutions-Handbook**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at **www.bpbonline.com** and as a print book

customer, you are entitled to a discount on the eBook copy. Get in touch with us at: **business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com.** We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com.**

Table of Contents

1. Installing Arduino IDE
- Introduction
- Structure
- Objectives
- Recipe 1: Arduino and its usage
  - *Advantages of Arduino board*
  - *Different Arduino boards*
  - *Features of Arduino Uno R3*
- Recipe 2: Installing Arduino IDE
  - *Offline IDE Installation for desktop environments*
- Recipe 3: Connecting Arduino board to the computer
- Recipe 4: Uploading a sample code on the Arduino board
  - *Troubleshooting in the uploading process*
  - *Compilation-related errors*
  - *Uploading related errors*
- Recipe 5: Adding and accessing a new library in Arduino sketch
  - *Accessing a pre-installed library*
  - *Installing new libraries from the library manager*
  - *Installing a library from a .zip file*
- Conclusion
- Points to remember
- Multiple choice questions
  - *Answers*

2. C Programming Basic
- Introduction
- Structure
- Objectives
- Recipe 1: Creating and saving your first Arduino sketch
  - *Structuring of an Arduino sketch*
  - *Writing the first Arduino sketch*

Recipe 2: Understanding different functions of Arduino programming
Recipe 3: Understanding variables in programming
Constants
Boolean constants (true and false)
Pin Levels: HIGH and LOW
Data types
Recipe 4: Display on serial monitor
Setting up serial communication and printing on the serial monitor
Recipe 5: Perform simple arithmetic operations in a sketch
A simple addition
More operations
Recipe 6: Accessing different types of pins on Arduino
Power, digital and analog
PWM pins
Serial communication pins
SPI and I2C pins
Interrupt pins
Recipe 7: Convert a number into a string
Conclusion
Points to remember
Multiple choice questions
Answers

3. Advanced Programming Construct
Introduction
Structure
Objectives
Recipe 1: Performing different actions based on two different variable values
Recipe 2: Performing multiple actions based on multiple values of a variable
Recipe 3: Repeating an action a fixed number of times
Recipe 4: Repeating an action till another event occurs
Recipe 5: Grouping and splitting different variables
Recipe 6: Splitting text from a group

Recipe 7: Interrupting an ongoing event with another event
Recipe 8: Generating different tones using Arduino uno
Recipe 9: Changing the intensity of an LED
Recipe 10: Sending binary values through serial communication
Recipe 11: Taking multiple decisions using case statements
Recipe 12: Generate the knight rider sequence using 7 LEDs
Recipe 13: Controlling 5 different LEDs using serial communication through PC
Conclusion
Points to remember
Multiple choice questions
Answers

4. Switches and Displays
Introduction
Structure
Objectives
Recipe 1: Reading different voltage values from a switch
Recipe 2: Calculating the time for which a pushbutton is pressed
Recipe 3: Generating red, green, and blue colors using an RGB LED
Recipe 4: Generating millions of random colors using RGB LED
Recipe 5: Driving a 7-segment LED display
Recipe 6: Displaying a blinking message on a 16x2 LCD
Recipe 7: Displaying a left scrolling message on a 16x2 LCD
Recipe 8: Displaying a rotating sequence using a NeoPixel LED
Recipe 9: Generating different voltage values using a potentiometer and display them on an LCD
Recipe 10: Design a 6-level water level indicator using an LED array
Conclusion
Points to remember
Multiple choice questions
Answers

5. Sensor Integration With Arduino
Introduction

Structure
Objectives
Recipe 1: Detecting the intensity of light
Recipe 2: Detecting the object with an IR proximity sensor
Recipe 3: Measuring the distance of a wall from the sensor
Recipe 4: Clap-controlled home appliance
Recipe 5: Designing a water level indicator by measuring the depth of the water level
Recipe 6: Detecting the LPG leakage in a home
Recipe 7: Detect the motion of a person around your home
Recipe 8: Design a robust burglar alarm system
Recipe 9: Securing access to a door using a 4-digit password
Recipe 10: Designing a multi-password door lock
Recipe 11: Designing a reverse car parking assistant system
Recipe 12: Measuring the temperature of the surroundings
Recipe 13: Designing a digital weather station to show real-time weather conditions
Conclusion
Points to remember
Multiple Choice Questions
Answers

6. Motor Control Using Arduino
Introduction
Structure
Objectives
Recipe 1: Connecting DC hobby motors to an Arduino
Recipe 2: Controlling the direction of a hobby motor
Recipe 3: Driving a 5 Volt DC motor using Arduino with L293D
Recipe 4: Controlling the speed of a motor using PWM
Recipe 5: Driving a 12 Volts DC motor using Arduino
Recipe 6: Designing a 2-wheel drive (2WD) robot
Recipe 7: Designing an obstacle avoider robot with Arduino
Recipe 8: Designing a line following robot to run on a black line
Recipe 9: Controlling the angle of movement of a servo motor

Recipe 10: Controlling the movements of a servo with a potentiometer
Recipe 11: A smart dustbin that opens the lid on detecting a person around it
Recipe 12: Automated parking boom barrier
Recipe 13: Designing a robotic hand to pick a glass of water
Recipe 14: Controlling a stepper motor with Arduino
Recipe 15: Controlling a DC motor with an encoder with Arduino
Conclusion
Points to remember
Multiple choice questions
Answers

7. I2C and SPI Communication
Introduction
Structure
Objectives
Recipe 1: Connect two Arduino boards to set up I2C communication
Recipe 2: Connecting I2C adapter to Arduino
Recipe 3: Controlling a 16x2 LCD with I2C protocol
Recipe 4: Displaying data on an OLED screen
Recipe 5: Digital voltmeter design using Arduino
Recipe 6: Understanding SPI communication on an Arduino board
Recipe 7: Communicate between two Arduino boards using SPI communication
Conclusion
Points to remember
Multiple choice questions
Answers

8. CAN Bus Communication
Introduction
Structure
Objectives
Recipe 1: Using CAN protocol with Arduino
Recipe 2: Setting up the MCP2515 module with an Arduino

Recipe 3: Send temperature sensor data over the CAN
Conclusion
Points to remember
Multiple choice questions
Answers

9. Bluetooth Communication With Arduino
Introduction
Structure
Objectives
Recipe 1: Understanding different Bluetooth modules to connect with an Arduino
Types of Bluetooth modules
Recipe 2: Establishing a Bluetooth connection between a PC and a Bluetooth module
Controlling HC-06 with AT commands
Connecting a Bluetooth module to the Windows PC
Recipe 3: Configure your smartphone to connect with Arduino via Bluetooth
Recipe 4: Visualize serial monitor information over Bluetooth
Recipe 5: Smartphone controlled car
Recipe 6: Bluetooth controlled servo motor
Conclusion
Points to remember
Multiple choice questions
Answers

10. Wi-Fi Connection Using Arduino
Introduction
Structure
Objectives
Recipe 1: Different options to connect to Wi-Fi via microcontrollers
Setting Arduino IDE for ESP32 or ESP8266 boards
Recipe 2: LED blink using ESP8266
Recipe 3: Connecting an ESP8266 to Wi-Fi

Recipe 4: Logging real-time light intensity values on the web
Recipe 5: Real-time Temperature and Humidity value logging on the web portal
Conclusion
Points to remember
Multiple choice questions
Answers:

Index

CHAPTER 1

Installing Arduino IDE

Introduction

The world around us is changing rapidly. We are witnessing a lot of automation in almost every field of work. A few decades ago, we had washing machines that worked as a mechanical assembly to wash the clothes for a given time. Over time, more features were added to these machines. You can control different washing modes, automatic start and stop options, and even control them through your smartphone via Wi-Fi. All these advancements are being developed using different microcontrollers. Most other home appliances, such as air conditioners, microwave ovens, vacuum cleaners, and refrigerators, are now being controlled by microcontrollers. Arduino is one of the most popular microcontrollers across the globe. This book will teach interesting recipes to make simple and complex systems using Arduino boards.

In this chapter, we will learn the basics of the Arduino UNO board through various recipes. Arduino UNO is not the only board developed by the Arduino project. There are several others that are developed for similar applications. We will discuss the reasons for selecting Arduino UNO for our learning. We will also learn the process to install the required software for programming an Arduino board. Once the **Integrated Development Environment** (**IDE**) is ready, we will learn how to connect an Arduino board to our computer. Towards the end of the chapter, we will learn the process of adding more libraries to the Arduino program.

Structure

In this chapter, we will discuss the following topics:

- Arduino and its usage
- Installing Arduino IDE
- Connecting Arduino board to a computer
- Uploading a sample code on the Arduino board

- Adding and accessing a new library in Arduino sketch
- Installing new **libraries** from the library manager

Objectives

By the end of this chapter, you will be able to do the following:

- Know about different Arduino boards and their usage
- Download and install Arduino IDE on any OS
- Connection and detection of an Arduino board with a computer
- Installing and accessing external libraries through IDE

Recipe 1: Arduino and its usage

Arduino is an open-source platform developed to support fast prototyping. The board is intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments. The board was first introduced in 2005 in Ivrea, Italy, by Massimo Banzi. The idea behind introducing this board was to provide a less expensive device for controlling interactive electronic projects and fast prototyping.

An Arduino board consists of an Atmel 8-bit AVR microcontroller. One of the most popular versions of the board, Arduino Uno, is shown in *Figure 1.1*. The board can sense the environment by receiving input from various sensors and actuators. The board has different types of pins to take different types of inputs and generate outputs. Since it is open-source hardware, these boards can be assembled or purchased preassembled. The Arduino programming language has a C-like syntax.

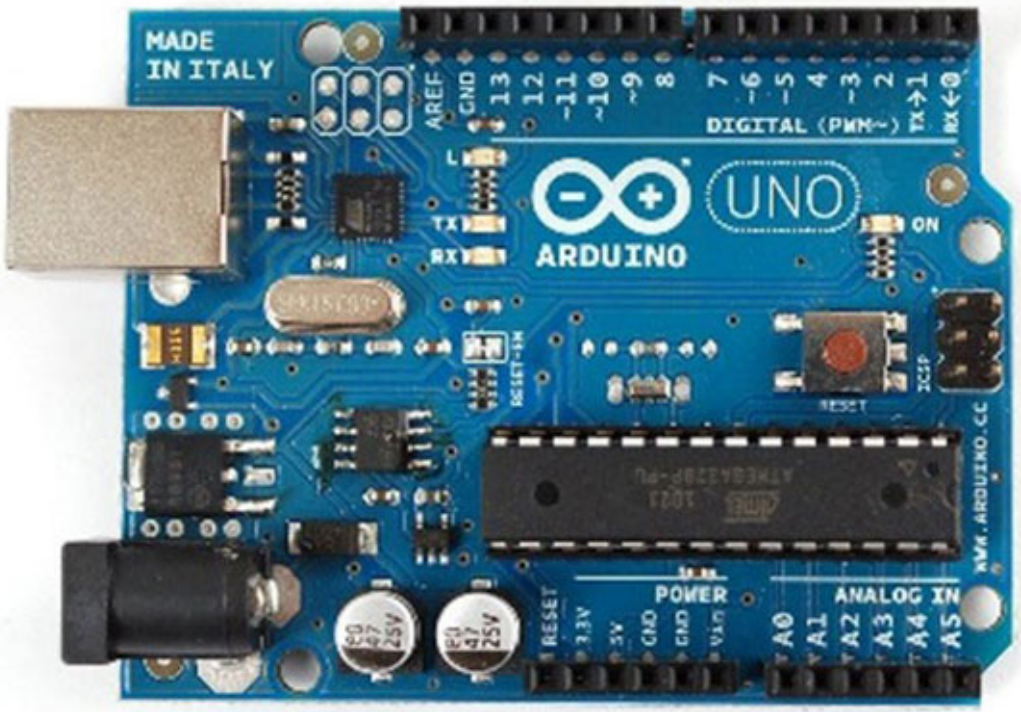


Figure 1.1: *Arduino Uno board*

Advantages of Arduino board

Arduino boards have gained popularity in the last decades and left every similar processing board far behind in the competition. The major advantages that this family of the board provides are as follows:

- **Inexpensive:** Arduino is an open-source project. Thus, it can be easily replicated and manufactured at a mass level. The original Arduino can cost around USD 25, while the copies are available for around USD 10.
- **Cross-platform:** Arduino supports programming and connections with almost every operating system for desktop and mobile platforms. It can be programmed using Windows, Linux, macOS, Android and iOS-based devices.
- **Simple, clear programming environment:** One of the reasons for the popularity of Arduino is the simple C++ based coding environment. The Integrated Development Environment (IDE) comes with a very clean

environment and already has the support of various libraries and in-built functions for programming.

- **Open source and extensible software:** The open-source software allows anyone to add libraries and functions to the existing list of libraries. So, if a company launches a new sensor, it can develop the relevant library and share it with the users.
- **Open source and extensible hardware:** Arduino is open-source, implying anyone can clone the board. It also implies that someone can extend the features and develop an extension board. A variety of drivers and shields are available to enhance the features of existing Arduino boards.

Different Arduino boards

The Arduino project is evolving, and the team regularly introduces new feature-rich boards. These boards are designed to fulfill different requirements of different user groups. Some applications are computation centric, while a few are networking-centric. We may process more analogue inputs in certain applications and digital inputs in others. In, *Figure 1.2*, we have shown the current active boards supported by the Arduino project. The list is available at **https://www.arduino.cc/en/hardware#boards-1**. There is a long list of retired boards as well. You can get to know about such boards at **https://www.arduino.cc/en/hardware#retired**.

By observing the wide variety of boards available, users can naturally get confused about the board to pick for their projects. The decision is made based on the requirement of the user. The best way to select the Arduino board is to check I/O and computing requirements. Each board varies based on the inputs, outputs, speed, form factor, voltage, and so on. The voltage required to operate these boards range from 3.7V to 5V:

Figure 1.2: *Different Arduino boards existing in the market today*

Features of Arduino Uno R3

Arduino Uno R3 is the best choice for beginning the journey with Arduino boards. It is one of the most robust boards to test your projects. Some of the key features of Arduino UNO are as follows:

- The board is based on the ATmega328P microcontroller.
- This board operates at 5V, but it can be driven by a DC voltage between 7 to 12 volts.
- Arduino UNO supports 14 digital input/output pics, 6 analogue input pins, and 6 Pulse Width Modulation (PWM) pins.
- Onboard pins are used to handle different types of input and output signals.
- Typically, the output current from these pins is around 20mA, which is good enough to drive the displays but insufficient to drive heavy motors.
- In addition to the basic features, Arduino Uno R3 supports 32KB of SRAM and 1KB of EEPROM, and it operates at a clock speed of 8/16 MHz.

Recipe 2: Installing Arduino IDE

Arduino board is a hardware, and we can program it with our specific commands and statements. To program any hardware, we need a software interface. This interface is known as **Integrated Development Environment** (**IDE**). Arduino IDE, or simply, Arduino software, provides the features of writing the code, checking the syntax, and uploading it to the hardware. Arduino IDE is available for offline as well as online coding experiences. For offline coding, you need to install the IDE once on your system, and then you do not have the dependency on an active internet connection. On the other hand, for the online editor, you need an active internet connection and a free account on the Arduino website to save your codes. The online code editor is available at **https://create.arduino.cc/editor**. You can create a free account using your Google, GitHub, Facebook, or Apple ID to sign in to the editor.

Offline IDE Installation for desktop environments

One of the most popular and more reliable methods of programming your Arduino board is using an offline IDE installed on your system. Arduino IDE is supported for almost every desktop and mobile OS. We will follow the given steps for installing Arduino IDE on Windows 10/11, Linus or macOS:

1. Go to **https://www.arduino.cc/en/software**
2. Scroll down a bit to the download section. The various options for respective operating systems are shown in *Figure 1.3*. The version of the IDE can be updated with time. So, you should download the latest one. You can always choose to download a previous version by scrolling down the page and finding your desired version. Refer to the following figure:

Figure 1.3: *Download options for Arduino IDE for different operating systems*

3. Click on the relevant link based on the OS, and you will be redirected to the download page. If you want to donate to the project, you can do that. Else, click on "**`Just download`**", as shown in *Figure 1.4*. The installer file will be saved to your system. Please refer to the following figure:

Figure 1.4: *Download page for .exe file required for Windows OS*

4. Sometimes, there might be no option to "**`Just Download`**", for one or more reasons. In that case, you should search for "**`Arduino IDE`**

`download`" on any search engine (Google, Bing, etc.). You will get the direct link for the installer from the official website.

5. The further steps show the installation for Windows OS. A similar set of steps can also be used for other operations systems. *Figure 1.5* and *Figure 1.6* show the steps involved in the installation process: Please refer to the following figure:

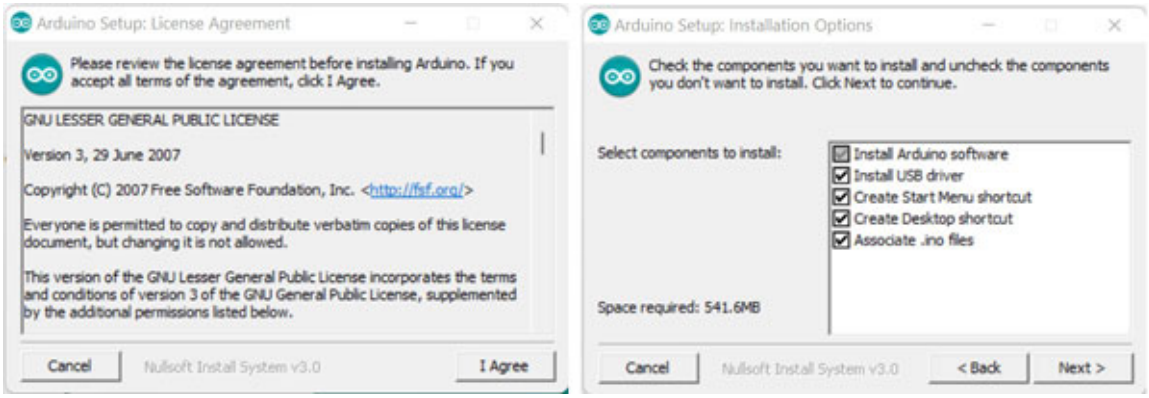


Figure 1.5: *Installing Arduino IDE on Windows OS*

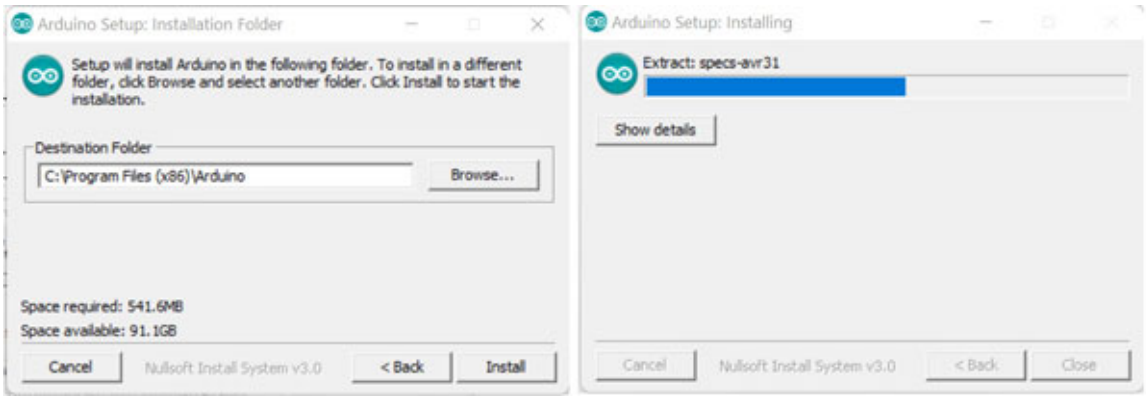


Figure 1.6: *Choosing the installation path and completing the IDE installation process*

6. The installer will also ask you to install some additional libraries, like "**Adafruit Industries LLC Ports (COM and LPT)**" and "**USB drivers**". You must also install these for a faster and more efficient connection of the Arduino board to the system. *Figure 1.7* shows the installation of cable (USB) drivers):

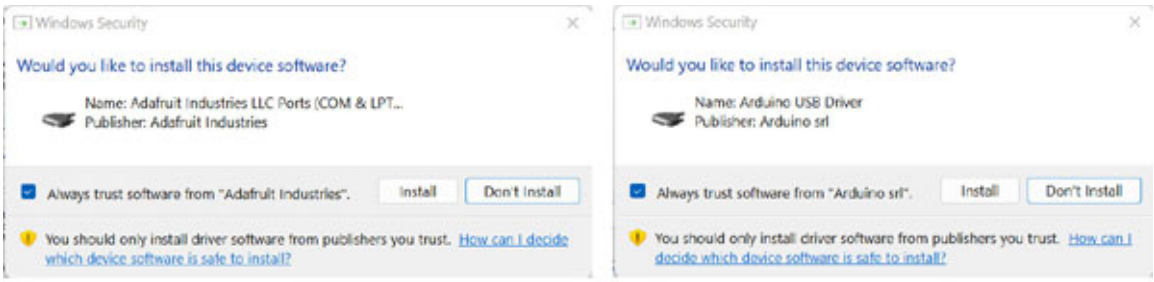


Figure 1.7: *Installing the USB drivers for efficient board connection*

7. The installation will be complete after this step.

You can follow similar steps for other operating systems as well. The interface for the Arduino IDE is shown in *Figure 1.8*. We get all the options to open, save, and close an Arduino sketch (In Arduino programming, a code is called a sketch.). We can edit the sketch and upload it using the same IDE. Please refer to the following figure:

Figure 1.8: *Arduino IDE interface*

Recipe 3: Connecting Arduino board to the computer

Once the software environment is ready, the next important step is to program our hardware (Arduino board) using the same platform. Connecting an Arduino board is quite an easy task. Still, we need to follow the given instructions carefully for proper connection and detection of the Arduino board to the system:

1. Connect the Arduino board with the USB cable provided along with it.

2. Refer to *Figure 1.9* to connect the board properly. The figure shows two different options to connect power to the Arduino board. We will select the USB port on the board. This port allows powering the board and setting up serial communication between the board and the computer. Please refer to the following figure:

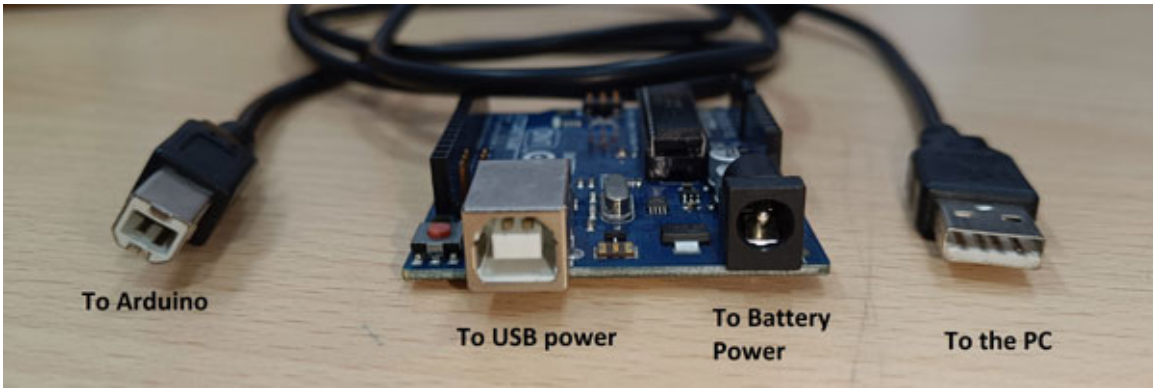


Figure 1.9: *Different power ports on the Arduino UNO and USB cable*

3. Connect one end of the USB cable to the Arduino board and the other to the USB port of your computer.
4. The "ON" Led on the board will glow when you connect the cable. This shows that the board is powered on.
5. Now, open the device manager on your computer; go to the ports section and find the port number of the Arduino board you have recently connected to. Remember the port number from the device manager.
6. *Figure 1.10* shows the detected port on the system. The Arduino board is detected at serial port COM4 in this case. It can be completely different for you. Every new board you connect to the system will connect to a new COM port.

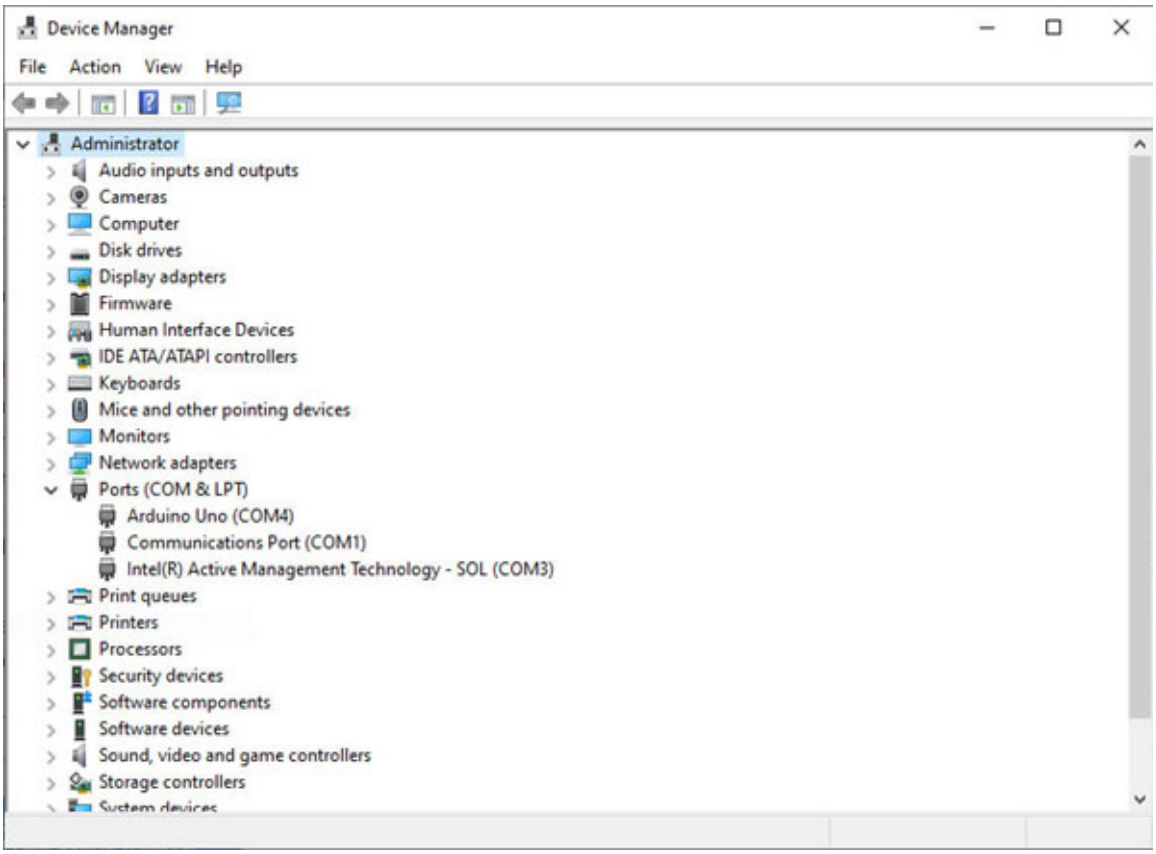


Figure 1.10: *Device manager showing that the Arduino UNO is connected to serial port COM4*

7. Go to **`Tools`** -> **`Board`** and make sure you select the correct hardware board. In this recipe, we are using Arduino UNO.
8. Go to **`Tools`**-> **`Port`** and select the port where your computer detects Arduino. In this recipe, we got **`COM4`** for Arduino UNO.
9. Once both board and port are selected properly, the process of connecting the Arduino board to the computer is completed.

Figure 1.11 shows the process of selecting the board and port values through the Tools menu:

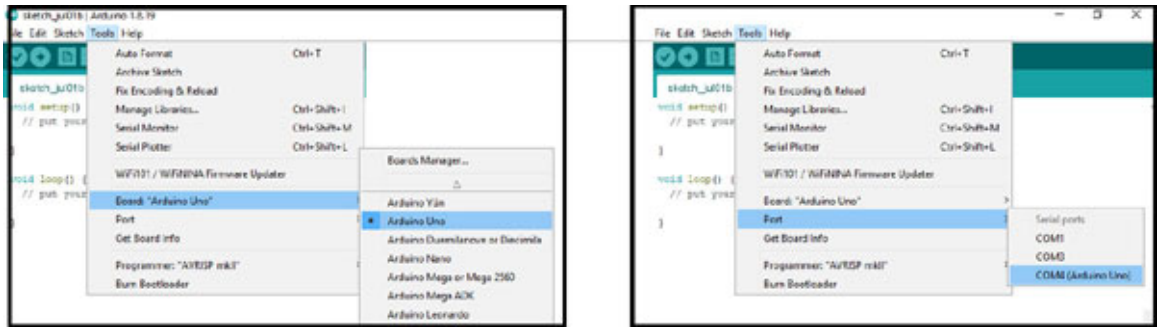


Figure 1.11: Setting up the correct board and port values via the Tools menu

Next time you plug your Arduino into the same computer, the IDE may independently set the board and port values. But verifying it every time you connect the board is always a good practice.

Recipe 4: Uploading a sample code on the Arduino board

This recipe will teach us how to upload an Arduino sketch to the Arduino board. Arduino sketches are saved with the **`.ino`** file extension. Since we have not learnt the Arduino programming till now, we will use an example sketch provided in the IDE and upload it on the board:

1. Go to Arduino IDE and open an example code from file -> **`Examples`** -> **`01. Basics`** -> **`Blink`**. It is shown in *Figure 1.12*:

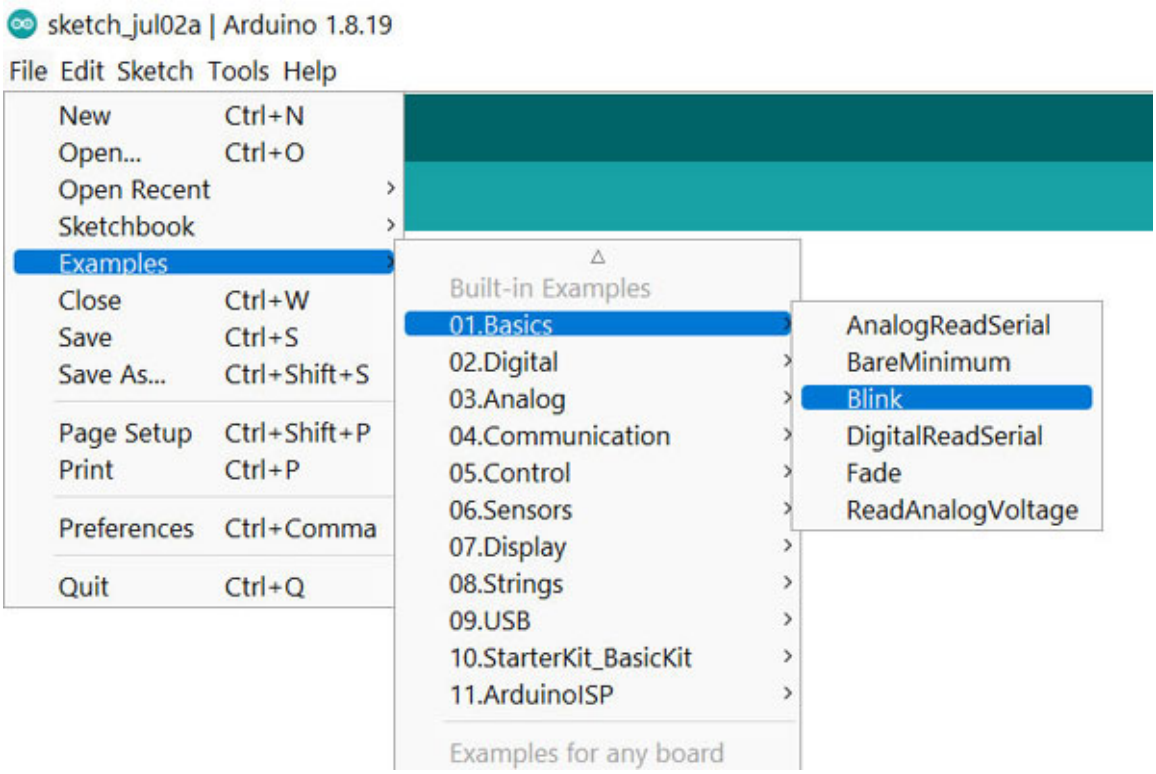


Figure 1.12: Opening an example sketch from in IDE

2. A new window will open with the code, which is as follows:

```
// the setup function runs once when you press reset or
power the board
void setup() {
```

```
// initialize digital pin LED_BUILTIN as an output.
pinMode(LED_BUILTIN, OUTPUT);
}
// the loop function runs over and over again forever
void loop() {
digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is
the voltage level)
delay(1000); // wait for a second
digitalWrite(LED_BUILTIN, LOW); // turn the LED off by
making the voltage LOW
delay(1000); // wait for a second
}
```

3. Connect the Arduino board to the computer by following the process explained in the previous recipe.
4. Go to the top-left corner of Arduino IDE and click on the verify button to check for any syntax errors in the sketch. The verify and upload buttons are shown in *Figure 1.13*:

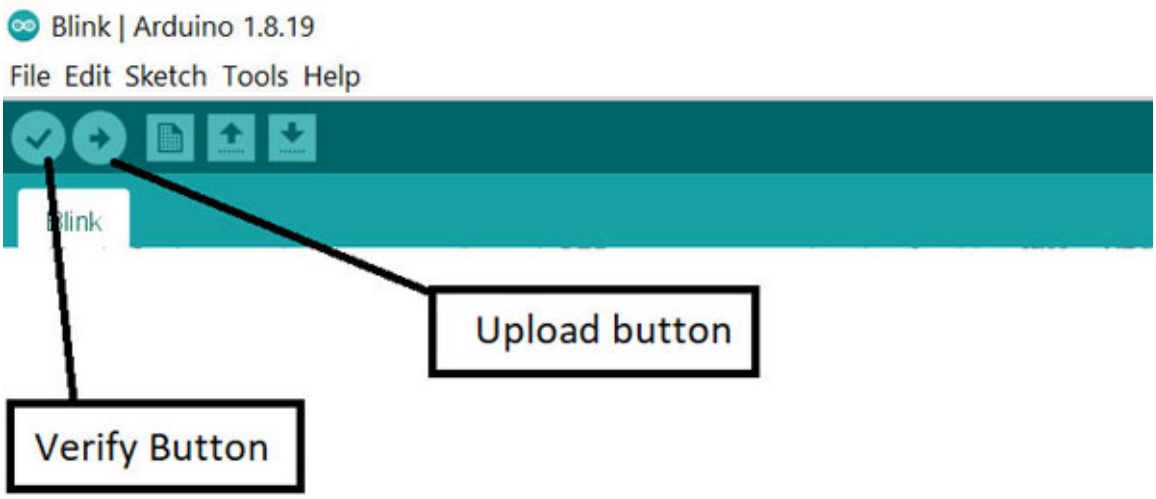


Figure 1.13: *Verily and upload buttons for syntax check and uploading*

5. If there are no errors in the sketch, then the bottom ribbon will show "**`Done compiling`**", as shown in *Figure 1.14*:

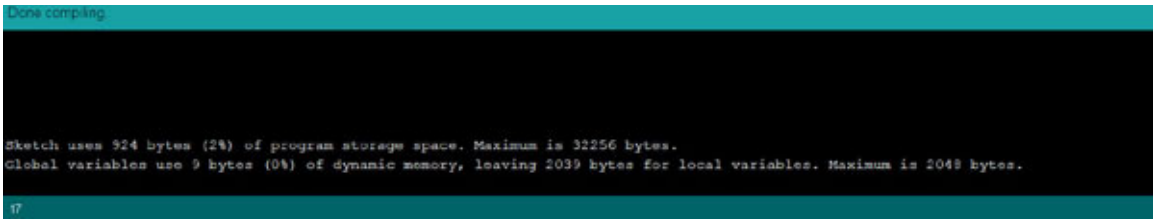


Figure 1.14: *A successful compilation of Arduino sketch*

6. Go to the top-left corner and click on the upload button. The compiled sketch image will be uploaded to the Arduino board. *Figure 1.15* shows the bottom ribbon status after successful uploading of the sketch.

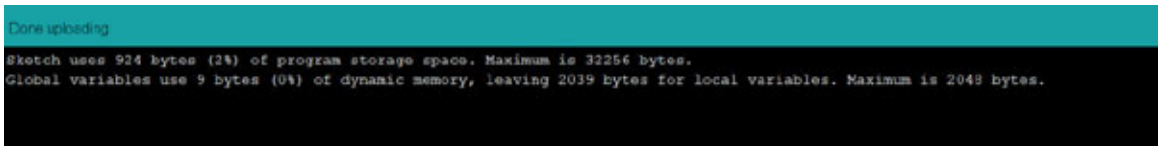


Figure 1.15: *Successfully uploading the sketch on an Arduino board*

Troubleshooting in the uploading process

Verifying the sketch and uploading the code can have some errors. Every such error is listed in the logs section of the IDE's bottom section. Most of the errors are self-explanatory and suggest possible solutions as well. When a sketch is compiled or uploaded, many logs are generated; by default, they are not visible in the Verbose (bottom ribbon). So, for any debugging, we must first enable verbose output to be shown for compilation and uploading.

Go to **`file`** -> **`Preferences`** and tick "*Show verbose output during compilation and upload*". *Figure 1.16* shows the preferences window. It would be best if you ticked both the compilation and upload options.

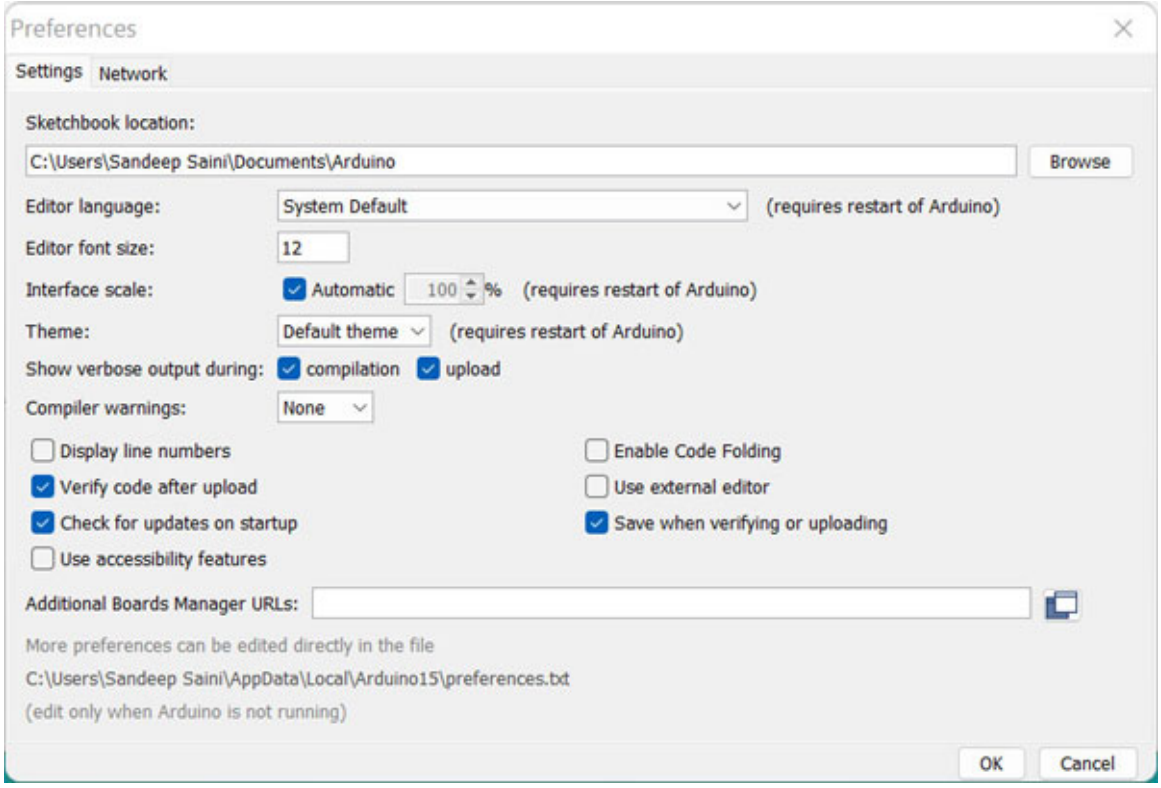


Figure 1.16: *Enabling verbose output for compilation and uploading operations*

Compilation-related errors

Most compilation-related errors are due to syntax errors in the sketch. For example, if something is wrong with the syntax, then one of the sample error messages is shown in *Figure 1.17*. The error message reads: **`Blink:35:3: error: expected ';' before 'digitalWrite'`**. Therefore, in line 35, the problem exists at the third character. The compiler cannot compile beyond this point, and it has returned the error message that before proceeding further, a semicolon (;) is missing. Thus, we go to the previous line and observe that a semicolon is missing after **`delay(1000)`** in line 34. Once we rectify the error and verify the sketch, it will compile without this error. Refer to the following figure:

```
25 // the setup function runs once when you press reset or power the board
26 void setup() {
27   // initialize digital pin LED_BUILTIN as an output.
28   pinMode(LED_BUILTIN, OUTPUT);
29 }
30
31 // the loop function runs over and over again forever
32 void loop() {
33   digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
34   delay(1000)                      // wait for a second
35   digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
36   delay(1000);                       // wait for a second
37 }
```

```
expected ';' before 'digitalWrite'

C:\Users\Sandeep Saini\Documents\Arduino\Blink\Blink.ino: In function 'void loop()':
Blink:35:3: error: expected ';' before 'digitalWrite'
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  ^~~~~~~~~~~~
exit status 1
expected ';' before 'digitalWrite'
```

Figure 1.17: *A simple compilation error in Arduino sketch*

We will discuss programming concepts and methods in the following chapters. We will also discuss how to debug these issues in detail.

Uploading related errors

One of the main causes for uploading related errors is the wrong connection of Arduino Uno. A loosely connected wire might not power the board. The second main reason for uploading related errors is the wrong selection of ports in IDE. Verifying the board type and port is always a good practice before uploading the sketch. *Figure 1.18* shows one such error where no board was detected at COM4:

```
Problem uploading to board. See https://support.arduino.cc/hc/en-us/sections/360003198300 for suggestions.

avrdude: ser_open(): can't open device "\\.\COM4": The system cannot find the file specified.


avrdude done.  Thank you.

Problem uploading to board.  See https://support.arduino.cc/hc/en-us/sections/360003198300 for suggestions.
```

Figure 1.18: *Error due to wrong port selection*

Recipe 5: Adding and accessing a new library in Arduino sketch

Libraries are sets of codes written for a specific set of applications. For example, if a company designs a new **Liquid Crystal Display** (**LCD**) or sensor, it provides the required codes to integrate hardware with the Arduino

IDE. A good number of libraries are inbuilt into the Arduino IDE. To access them, we can use **Library Manager** in the IDE.

In this recipe, we will learn the following:

- Accessing a pre-installed library
- Installing a new library from the library manager
- Installing a new library from a ZIP file

Accessing a pre-installed library

Let us learn the simple process of accessing a new library through these simple steps:

1. Go to `Sketch` -> `Include library`
2. You can check the list of pre-installed libraries in the list. *Figure 1.19* shows the default libraries available with IDE 1.8.19.
3. If we want to work with any of these existing libraries, we select the desired library, and the Arduino sketch will include the desired library. The same figure shows the inclusion of Liquid Crystal Library in the sketch by selecting it from the list of libraries:

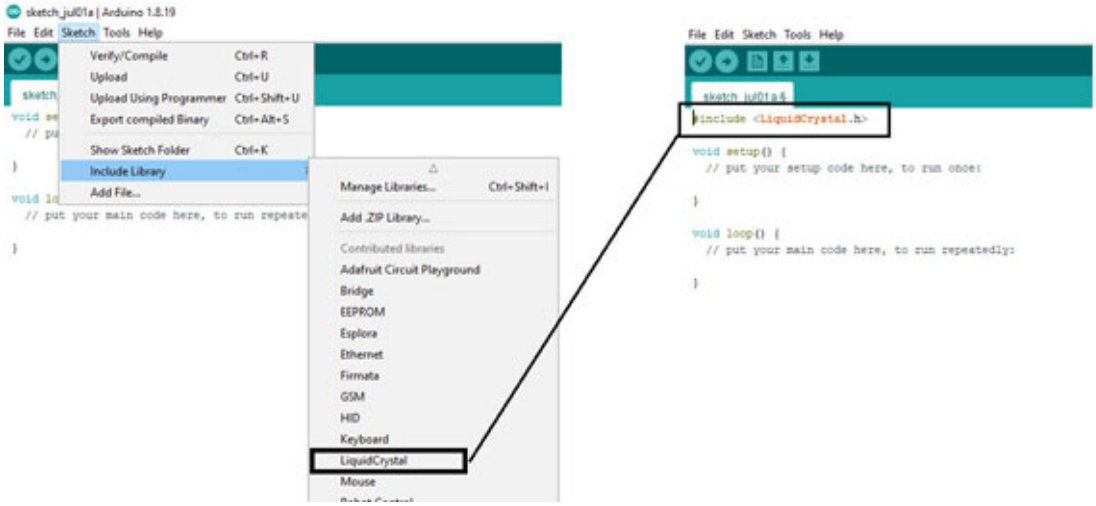


Figure 1.19: *List of pre-installed libraries on Arduino IDE and selecting one for the sketch*

Installing new libraries from the library manager

Sometimes, we use complex or rare components with Arduino board. These components have their own libraries that are designed by the manufacturer.

Such libraries are not a part of Arduino IDE by default. We can include such libraries in our IDE by following these steps:

1. Go to **`Sketch`** -> **`Include library`** -> **`Manage Libraries`**, or use the keyboard shortcut *Ctrl+Shift+I*
2. The library manager will open.
3. You can scroll down or use the search option to search for any desired library. *Figure 1.20* shows the library manager opened in Arduino IDE:

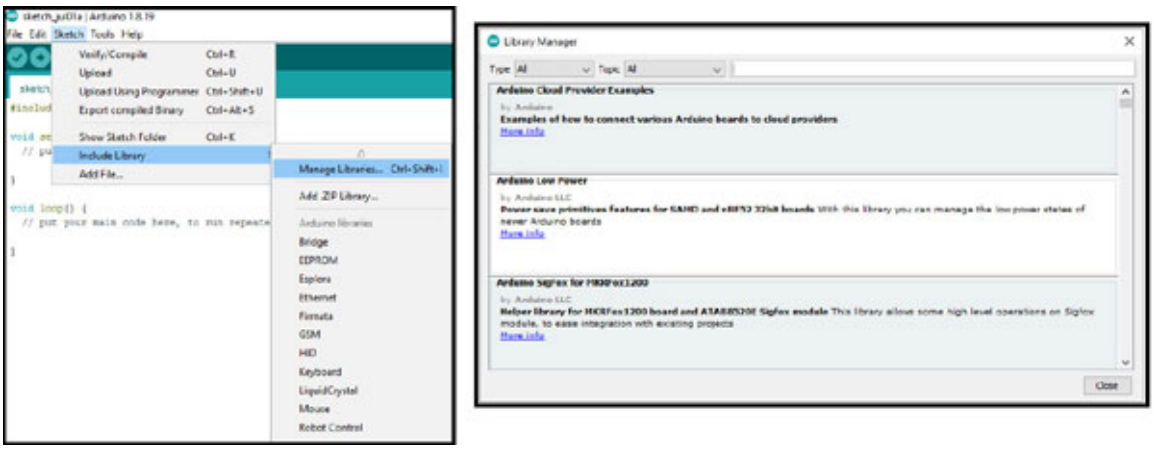


Figure 1.20: *Accessing and exploring the Library Manager from Arduino IDE*

4. In this example, we will search for Adafruit Feather M4 CAN board CANFD driver. The search results are displayed.
5. You can select the version of drivers and click on install.
6. The libraries will be installed and added to the list of pre-installed libraries.
7. *Figure 1.21* shows the process of searching for a new library and how it will be displayed in the list of existing libraries.

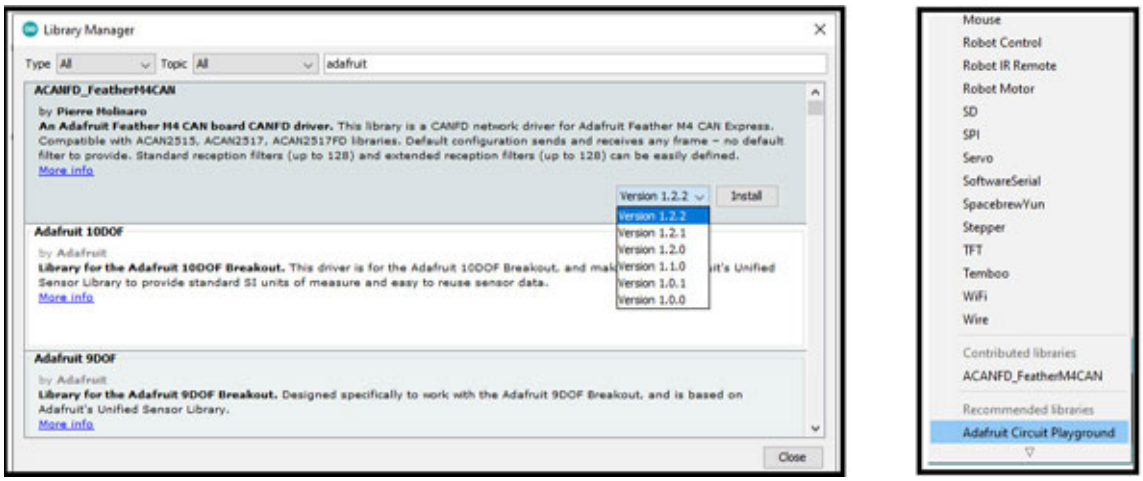


Figure 1.21: *The process of searching for a new library and installing a specific version on the system*

Installing a library from a .zip file

Sometimes, when you visit the home page of any manufacturer's website, they provide software support for their products. The relevant libraries are provided as archive (.zip) files. We can also install the library package in the Arduino IDE using a .zip file:

1. Download the .zip file from the resource section and save it to your system.
2. Go to **`Sketch`** -> **`Include library`** -> **`Add .ZIP Library`**.
3. Browse the .zip file through explorer and open it.
4. The library is added to the IDE.

Figure 1.22 shows the process of adding the library "**`OLED Display 96x96`**" through a .zip file:

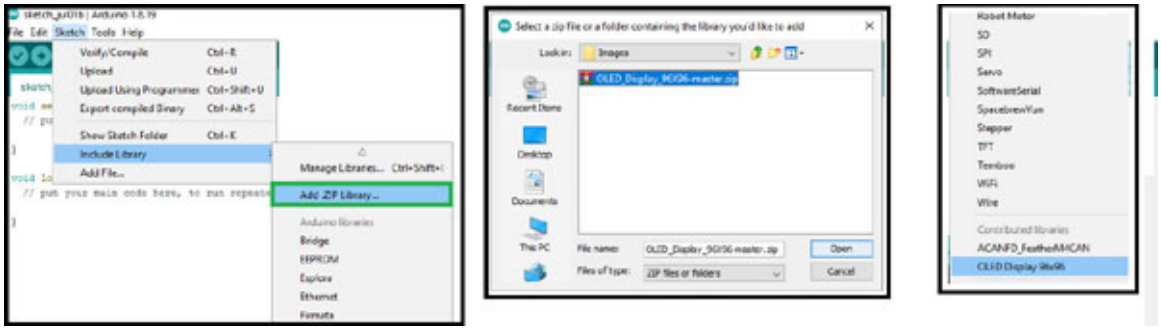


Figure 1.22: *Adding library through .zip files to the IDE*

Conclusion

In this chapter, we gained information about various boards developed under Arduino project. We discussed the reasons for selecting Arduino UNO as our first prototyping board. We also learned about various features of this board and different pins for taking inputs or providing outputs. We understood how to install the IDE for programming an Arduino board. Then, we talked about connecting an Arduino board to the computer and detecting it on IDE through correct board and port combination. The board was ready for use after successful integration. Further on, we learned about adding and accessing more libraries to the core IDE.

Once we have the basic connections ready, the next task is to program the Arduino board. In the next chapter, we will start learning the basics of Arduino C language through various recipes. These recipes are designed in a way that the basic concepts will be covered first. And, as we go ahead, we will introduce complex programming constructs through these recipes.

Points to remember

In this chapter, we mainly discussed the process of integrating the Arduino board to our computer. Always remember the following points to make sure your connections are right:

- Download and install the latest stable version of IDE for your operating system.
- Make sure the power on LED of Arduino is on when you connect it to the PC.
- Always select the correct board on IDE.
- Always select the right port number for code uploading on Arduino.
- Make sure the desired libraries are installed in your IDE.

Multiple choice questions

1. **Arduino Uno is having ____ digital pins.**

    a. 12

    b. 14

c. 6
d. 8

2. **Which feature is not available on Arduino UNO?**
   a. Support to take digital inputs
   b. 12C and SPI communication
   c. Wi-Fi communication
   d. Serial communication
3. **Arduino IDE can provide access to external libraries through _____.**
   a. Serial monitor
   b. Library monitor
   c. Library master
   d. Library manager
4. **It is not possible to download a library in .zip format and integrate it with Arduino IDE.**
   a. False
   b. True
5. **What is the operating voltage for Arduino UNO?**
   a. 5V
   b. 3.3V
   c. 7.4V
   d. 12V

Answers

1. **b**
2. **c**
3. **d**
4. **a**
5. **a**

CHAPTER 2

C Programming Basic

Introduction

In the previous chapter, we discussed the software and hardware integration for connecting an Arduino board to the computer. The next task is programming the board with user-defined commands. This section will have two chapters dedicated to the programming concepts. We will learn the C programming basics to program the Arduino board with our desired commands.

This chapter will teach coding from scratch with a series of recipes. Each recipe will introduce a new feature of programming for the Arduino board. An Arduino program is called a sketch. We will discuss the different components of an Arduino Sketch and methods of programming an Arduino in the beginning. Every Arduino board has different types of pins, and we will learn how to access data from these different pins. After the basic knowledge of pins and sketch structure, we will write out the first program and test it on an Arduino board. This chapter will also introduce the process of accessing a serial monitor.

Structure

In this chapter, we are going to discuss the following recipes:

- Creating and saving your first Arduino sketch
- Understanding and accessing different datatypes in programming
- Understanding variables in programming
- Display on serial monitor
- Performing simple arithmetic operations in a sketch
- Accessing different types of pins on Arduino
- Converting a number into a string

Objectives

By the end of this chapter, the learner will be able to do the following:

- Open, create and save an Arduino sketch using IDE
- Identify and use different data types in Arduino programming
- Create and use variables in programming
- Access and use the serial monitor to display the output from the Arduino board
- Access different types of pins on an Arduino board

Recipe 1: Creating and saving your first Arduino sketch

So, we have our Arduino board, and we want to start with it now. We uploaded an example sketch to the Arduino board in the previous chapter. How about writing our own sketch now? In this recipe, we will learn the basic structure of an Arduino sketch and the process of writing our own sketch.

Structuring of an Arduino sketch

When we open IDE, then a blank sketch is opened in the editor. We can use file options to save this sketch at our preferred location on the computer. By default, the name of the sketch is derived from the date and saved in the **Documents** folder of your Windows OS. It is always a good practice to save the sketch with an appropriate name before starting the programming. The name of the Arduino sketch can contain any string of letter, digit, and underscore characters only. Any other special character is not allowed. Also, the filename cannot start with a number. *Figure 2.1* shows an Arduino sketch saved with the name **my_first_sketch.ino**:

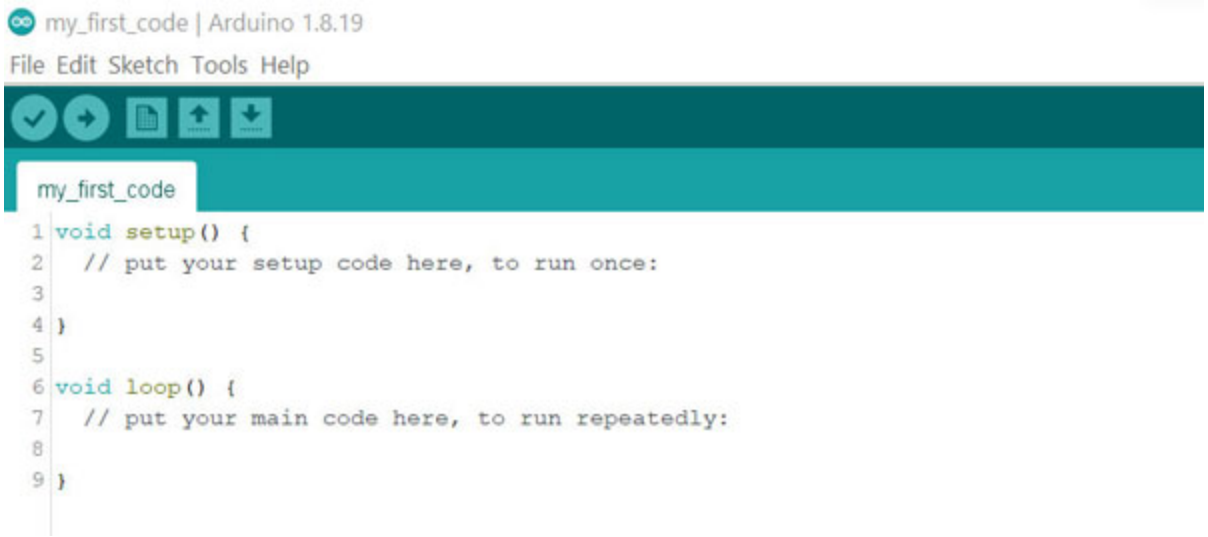


Figure 2.1: *A code saved with the file name my_first_sketch*

Any Arduino sketch must consist of two functions: **`setup()`** and **`loop()`**. The Arduino sketch can contain more functions as well. But these two are a must for a sketch to work properly. The sketch shown in *Figure 2.1* also shows these two functions. The **`setup()`** function is the first function in any Arduino sketch. It contains the initial settings for various pins and declarations of new variables (if required). We define whether we will use a particular pin as an input or output pin. The **`loop()`** function consists of the set of statements that will be executed forever. These statements will be executed in the loop until the board is powered on. The function can have multiple statements, and they are compiled sequentially. That means that if a certain statement is directed to another external function, the compiler will exit the **`loop()`** function and switch to the called function.

Comments in Arduino sketch are started with `//`. The compiler will not compile anything beyond `//` in that line.

Writing the first Arduino sketch

Once we know the basic structure of an Arduino sketch, let us write a simple code to blink at LED at different time intervals. In the example sketch, we did not pay much attention to the content of the sketch. Let us write similar code now and understand it in detail.

The setup function is written as follows:

```
void setup() {
  pinMode(13, OUTPUT); // setting pin no. 13 as output pin.
}
```

We have defined one statement in this function. The statement is setting pin number 13 as the output pin. Therefore, we cannot attach any input to this pin now.

We could have done the same differently as well:

```
int out_pin = 13;
void setup() {
  pinMode(out_pin, OUTPUT); // setting pin out_pin as output
  pin.
}
```

We initially declared a global variable named **`out_pin`** in this function and set its value to 13. Then, we set the pin number as our output pin equal to the variable value.

Now, let us write the code for the **`loop()`** function. We aim to glow an LED attached to pin number 13 for 5 seconds and then keep it off for 2 seconds. In Arduino sketch, we can provide delay by using the **`delay()`** function. The argument is given in milliseconds. So, if we want to provide a delay of 5 seconds, we use **`delay(5000)`**. The code for the **`loop()`** functions is as follows:

```
void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(13,HIGH);
  delay(5000);
  digitalWrite(13,LOW);
  delay(2000);
}
```

Recipe 2: Understanding different functions of Arduino programming

We learned in detail about the **`setup()`** and **`loop()`** functions in the previous recipe. Arduino programming language has many other predefined

functions to make our life easy in programming an Arduino. Some of these functions and their brief functionalities are mentioned in *Table 2.1*. We will be using these functions in many future recipes. These functions make programming easy for us as we do not have to write these functionalities again to use them in our sketch. For example, if we have to find the maximum of given n numbers, we can write a C program to compute the maximum of n integers. On the other hand, if we have a predefined function to find the maximum number, we do not have to put effort into writing the code for these functions. Some of the most common functions are as follows:

Category	Function	Description
Digital (I/O)	`digitalRead()`	Reads the input in digital form from a pin
	`digitalWrite()`	Writes the digital output value to a digital pin; only HIGH and LOW values can be written
	`pinMode()`	Sets the mode (input or output) of a pin to be used in the sketch
Analog (I/O)	`analogRead()`	Reads the analog input from an analog input pin
	`analogWrite()`	Writes the **Pulse Width Modulation** (**PWM**) output on a pin
Time	`delay()`	Provides delay in milliseconds
	`delayMicroseconds()`	Provides delay in microseconds
	`micros()`	Counts the number of microseconds passed from the start of the program execution
	`millis()`	Counts the number of milliseconds passed from the start of the program execution
Maths	`abs()`	Returns the absolute value of a number
	`constrain()`	Constrains a number to be within a range
	`map()`	Maps one range of numbers with another range
	`max()`	Returns the maximum of the given numbers
	`min()`	Returns the minimum of the given numbers
	`pow()`	Calculates the value of a number raised to a power
	`sq()`	Returns the square of a number

	`sqrt()`	Returns the square root of a number
Random numbers	`random()`	Generates a pseudo-random number between 0 to the given number
	`randomSeed()`	Initializes the pseudo-random number generator, causing it to start at an arbitrary point in its random sequence
Interrupts	`interrupts()`	Enables considering interrupts in the loop function
	`noInterrupts()`	Disables considering interrupts in the loop function

Table 2.1: *Predefined functions in Arduino Programming language*

Recipe 3: Understanding variables in programming

In the first recipe of this chapter, we defined a new variable called `out_pin` and fixed its value to 13. In this declaration, we used an integer datatype. So, we may have a question in our mind about different other types of variables and data types that are supported in Arduino programming. These variables and data types will be vital for our future recipes. You can consider them the ingredients for future recipes. And we must know the right set of ingredients for a perfect recipe.

Constants

Constants are used to make the program more readable and comprehendible, and they have fixed values. There are various groups of constants.

Boolean constants (true and false)

True is defined as logic 1. Any non-zero integer is also true.

False is used or logic 0. Both true and false are written in complete lower cases. Since C is a case-sensitive language, writing these keywords in proper cases is vital.

Pin Levels: HIGH and LOW

One of the most commonly used sets of pins on Arduino board is digital pins. These pins can be assigned two constant values:

HIGH: if the voltage level is more than 3.0 (for 5V supply) or 2V (for 3.3V supply), then the logical value of the pin is considered HIGH.

- **LOW:** If the voltage level is less than 1.5 (for 5V supply) or 1V (for 3.3V supply), then the logical value of the pin is considered LOW.

Data types

The Arduino board is capable of accepting, processing, and generating a variety of data types. Therefore, we must know about the different data types supported in the programming. *Table 2.2* shows the supported data types and their brief descriptions:

Data type	Brief description
array	An array is a set of multiple variables stored with index values.
bool	It is used for using values true ad false only.
byte	A byte stores an unsigned 8-bit number whose value can be between 0 to 255.
char	It is used to store characters. Single characters must be declared using single quotes and longer characters using double quotes. For example, ‘a’, “and”.
double	It is used to store double-precision floating-point numbers. It uses 4 bytes of memory.
float	Floating-point representation is used when a number has decimal points. It also uses 4 bytes.
int	This is one of the most commonly used data types. It stores integers in the range -32,768 to 32,767.
string	It is an array of characters. It can be an array of any length characters.

Table 2.2: *Data types supported for Arduino Programming*

Recipe 4: Display on serial monitor

From the previous two recipes, we have a fair idea of the different functions, variables and data types used in the programming. We expect to

verify the output when we write a sketch and run it on the Arduino board. Some of the outputs can be seen via display devices (LED, LCD, OLED), while a few can be observed via mechanical movements of actuators. However, most computations leading to these final outputs cannot be observed using these mechanisms.

In Arduino IDE, we have the support of a Serial monitor to print any values while executing the program. The serial monitor works on the principle of serial communication between the Arduino board and the computer via USB cable. Serial communication is two-way communication. This implies that we can print something from Arduino to the computer screen, and we can also provide inputs from the computer to the Arduino.

Go to **`Tool`**-> **`Serial monitor`** or use the shortcut *Ctrl+Shift+M* to open the serial monitor. *Figure 2.2* shows the serial monitor interface in IDE. It must be noted that the serial monitor can be accessed only if the Arduino board is connected to the computer and the port is selected correctly:

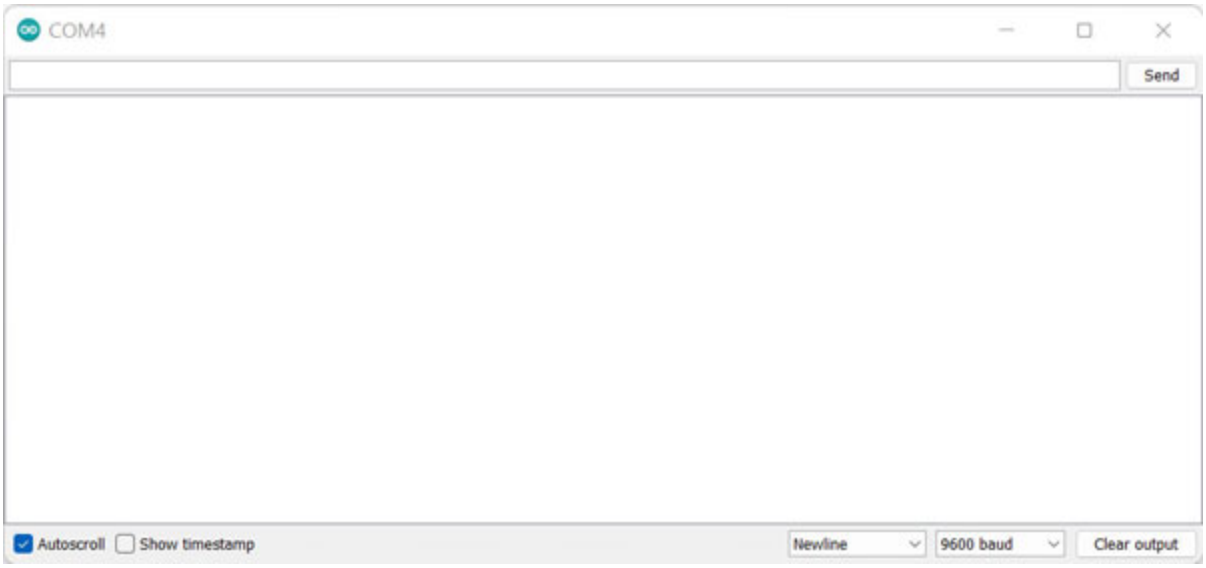


Figure 2.2: *The serial monitor interface in Arduino IDE*

We can observe two main sections in the serial monitor: the upper section is an input terminal, where we can send user commands to the Arduino board, and the lower section prints the output from the Arduino board. Currently, nothing is being shown on the screen as there is no serial communication set between the board and the computer.

Setting up serial communication and printing on the serial monitor

Serial communication works at a particular speed between two devices. Typically, we set the communication rate between a computer and Arduino board at 9600 bauds per second (bps). To print something from the Arduino board to the computer, we need to perform two operations:

Set the serial communication inside the **`setup()`** functions. Use the appropriate commands to print inside the **`loop()`** function.

A sample sketch to print "**`Hello World`**" is given as follows:

```
void setup() {
  Serial.begin(9600);
}
void loop() {
  Serial.print("Hello World");
  delay(1000);
}
```

This sketch will print **"`Hello World`"** on the serial monitor screen every second. The output of this sketch on the serial monitor is shown in *Figure 2.3*:

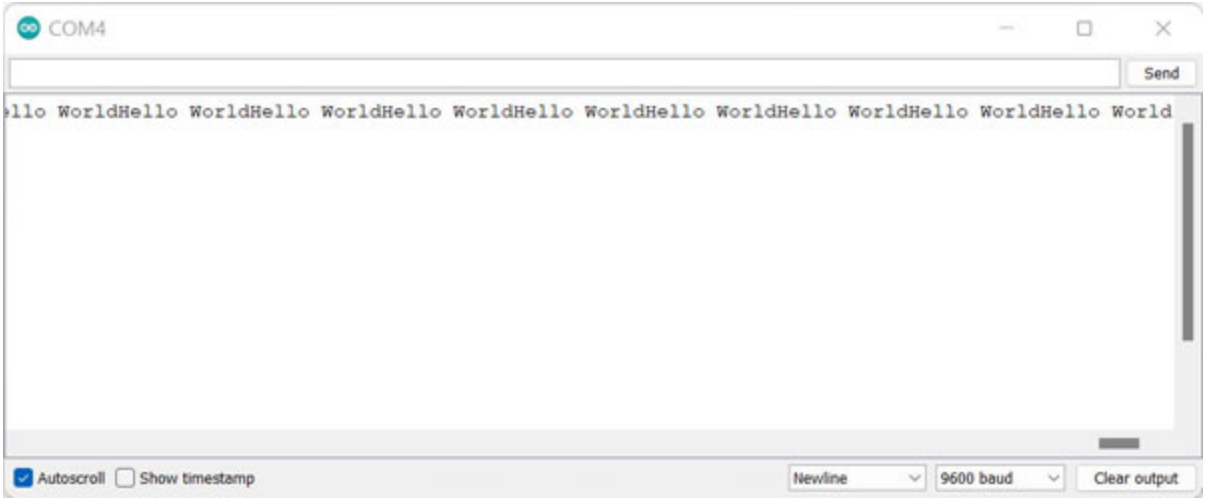


Figure 2.3: *Printing "Hello World" using serial communication*

It can be observed that the Tx LED on the Arduino board will also blink every second. This indicates that the board is transmitting the data serially

every second.

The second thing we observe is that the "**`Hello World`**" string is printed one after another. If we want to print one string per line, we use the **`Serial.println()`** function. The **`loop()`** function of the modified sketch is given as follows. *Figure 2.4* shows the printing of "**`Hello World`**" in new lines.

```
void loop() {
  Serial.println("Hello World");
  delay(1000);
}
```

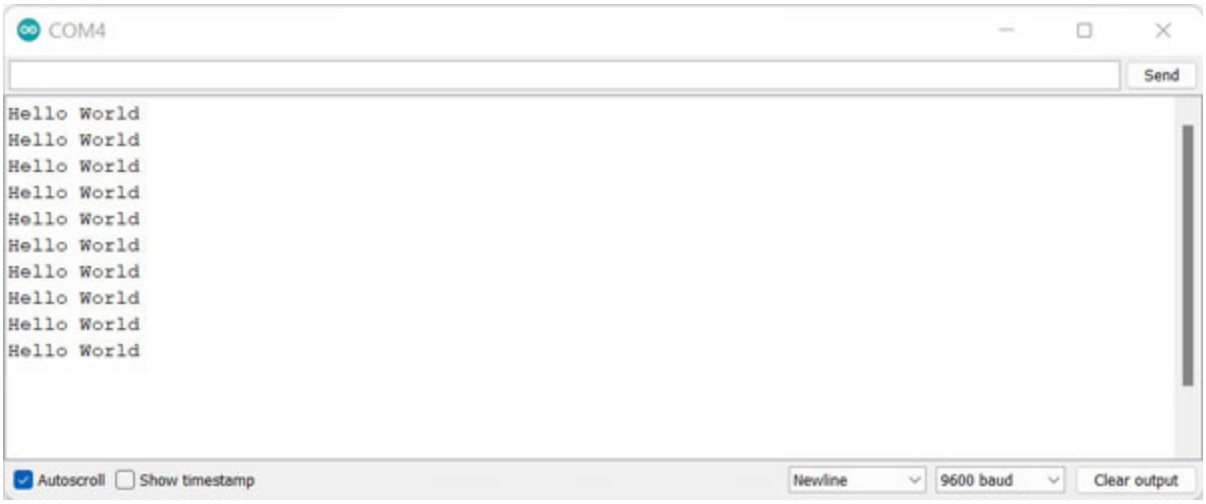


Figure 2.4: *Printing "Hello World" in new lines*

Recipe 5: Perform simple arithmetic operations in a sketch

We have good knowledge of some very basic programming constructs now. In this recipe, we will perform certain mathematical operations on some variables and print them on the serial monitor.

A simple addition

The following is a very simple sketch to add two numbers and print their sum on the serial monitor. We have declared two integer variables in this

sketch: **a** and **b**. We have also assigned them fixed values. Then, we have declared a new variable: **sum**.

These declarations are made outside any of the functions. It implies that these are global variables and will store the same values in any function until modified by a certain function. The output of this program is shown in *Figure 2.5*.

```
int a = 21;
int b = 38;
int sum;
void setup() {
 Serial.begin(9600);
}
void loop() {
 sum = a + b;
 Serial.println(sum);
 delay(1000);
}
```

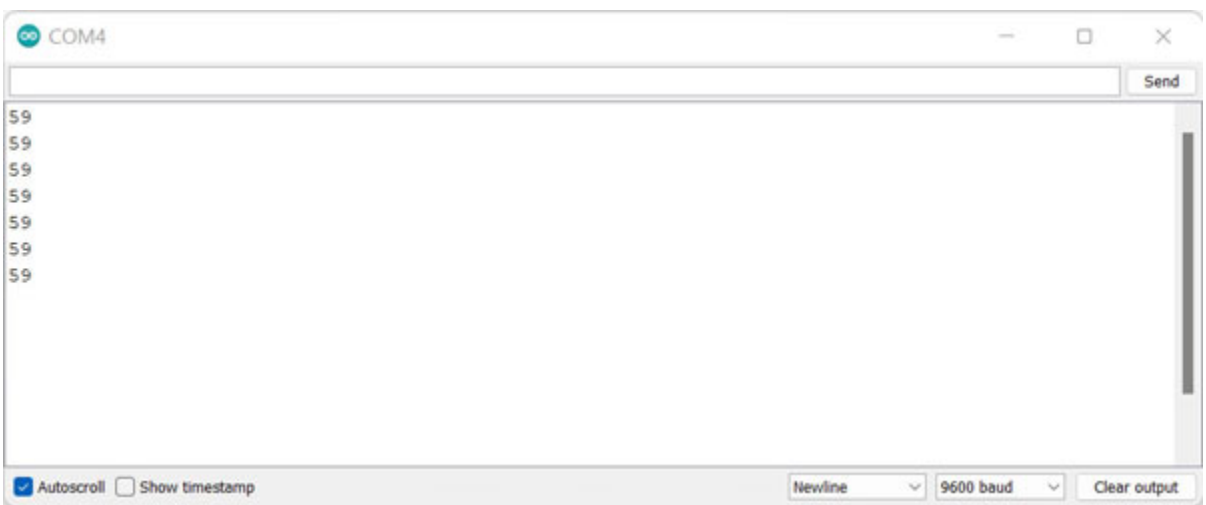


Figure 2.5: *Printing the sum of 21 and 38 on the serial monitor*

More operations

Let's perform a few more operations and understand how it works:

```
int a = 21;
int b = 38;
```

```
int c;
void setup() {
 Serial.begin(9600);
}

void loop() {
 c = a + b;
 Serial.print("The sum of a and b = ");
 Serial.println(c);
 c = a * b;
 Serial.print("The product of a and b = ");
 Serial.println(c);
 c = a/b;
 Serial.print("The quotient after dividing a by b = ");
 Serial.println(c);
 c = a%b;
 Serial.print("The remainder after dividing a by b = ");
 Serial.println(c);
 delay(1000);
}
```

The output of the sketch is

```
The sum of a and b = 59
The product of a and b = 798
The quotient after dividing a by b = 0
The remainder after dividing a by b = 21
```

Recipe 6: Accessing different types of pins on Arduino

Arduino UNO board contains different categories of pins. All these pins are used for different purposes. We will start comprehending each category of these pins in this recipe. Every pin on the board is given a unique number and name to identify them in the sketch. Arduino pins can be used for two main purposes. One takes inputs from external sources like sensors, and the other provides outputs to actuators and display devices. We use the **`pinMode()`** function to define any pin's input or output mode of operation.

This must be defined in the **`setup()`** function. The syntax for the **`pinMode()`** function is **`pinMode (pin, mode);`**. Here, the **`pin`** is the number of the pin whose mode you wish to set, and the **`mode`** can be **`INPUT`** or **`OUTPUT`**. A sample declaration is shown as follows:

```
void setup() {
  pinMode(13, OUTPUT);
  pinMode(A0, INPUT);
  pinMode(2, INPUT);
}
```

In this sketch, we have assigned digital pin 13 as an output pin, analog pin A0 as an input pin, and digital pin 2 as another input pin.

Power, digital and analog

Let us discuss different categories of pins on an Arduino board now. The first and foremost thing is to power the board or other components from the board. Then come the inputs and outputs. There can be digital and analog input outputs connected to an Arduino board. So, let us first consider these three categories and the process to access them.

Figure 2.6 shows the three primary categories of pins on an Arduino UNO board. Power pins are mainly used to power other components through the board. One of the most vital pins in this category is the ground (GND) pin. Even if we use external batteries to power other sensors and actuators, we must join both ground pins to one common point. If we do not merge the grounds of all the power sources, we get unreliable outputs. Refer to the following figure:

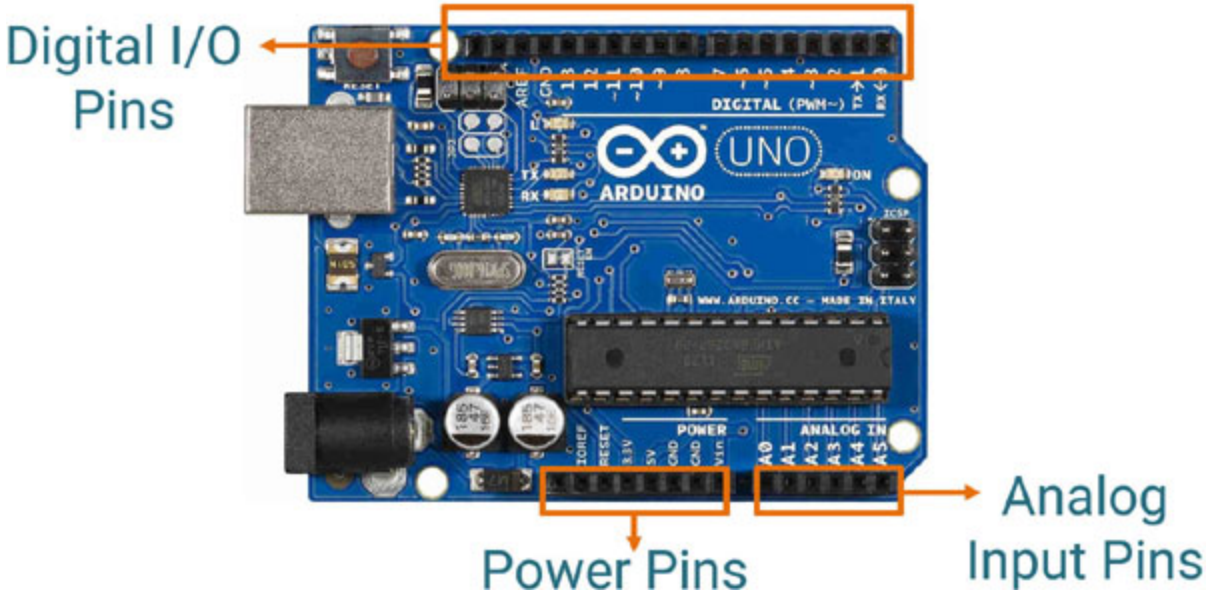


Figure 2.6: *Power, Digital I/O and Analog input pins on Arduino UNO*

We can do the following through power pins:

- Provide ground connection to other components
- Provide 5V supply to other components
- Provide 3.3V supply to other inputs

These pins are not programmable and should not be connected in input mode in any case.

Digital pins are one of the most vital and commonly used on the board. There are 14 digital input/output pins. These are numbered from 0 to 13. All of these are capable of reading digital input and writing digital outputs. However, in one sketch, one pin should be used in one mode (either input or output) only.

There are 6 analog pins on an Arduino UNO board. These are only input pins. These pins are used to read the analog voltages from 0 to 5 volts. These pins have a 10-bit **Analog to Digital Converter** (**ADC**) attached. Therefore, we can have 2^10 = 1024 different values of inputs received at these pins. That implies that every voltage in the range of 0 to 5 volts will be mapped to an integer in the range of 0 to 1023 (2^10 -1 = 1023). It is very important to know that the microcontroller on the board can only

process digital values and therefore, direct analog signals cannot be processed by the board.

The following sketch is an example that shows reading inputs from analog pins and writing some digital outputs on the digital pins:

```
void setup() {
  pinMode(13, OUTPUT);
  pinMode(A0, INPUT);
  pinMode(2, INPUT);
  Serial.begin(9600);
}
int a = 0;
int b = 0;
void loop() {
  a = analogRead(A0);
  Serial.print(“The value read at pin A0 = “);
  Serial.println(a);
  b = digitalRead(2);
  Serial.print(“The value read at pin 2 = “);
  Serial.println(b);
  digitalWrite(13, b);
  delay(1000);
}
```

The output of the sketch on the serial monitor is shown in *Figure 2.7*. We can observe that the outputs change as the analog pin values change. However, the output at digital pin 2 is stable as we are not changing it.

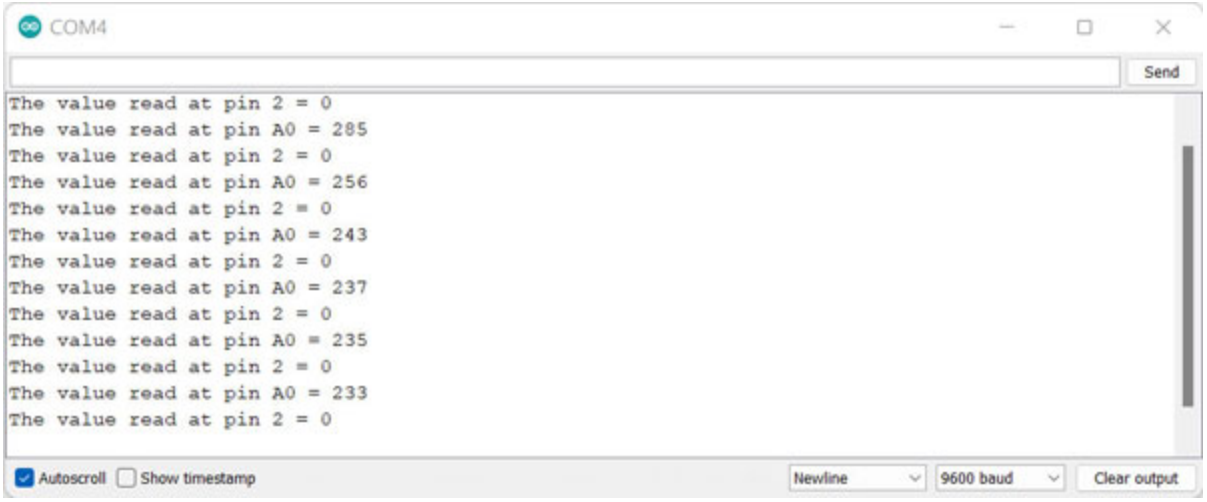


Figure 2.7: *Reading analog and digital values on respective pins*

The circuit connection for the same is shown in *Figure 2.8*:

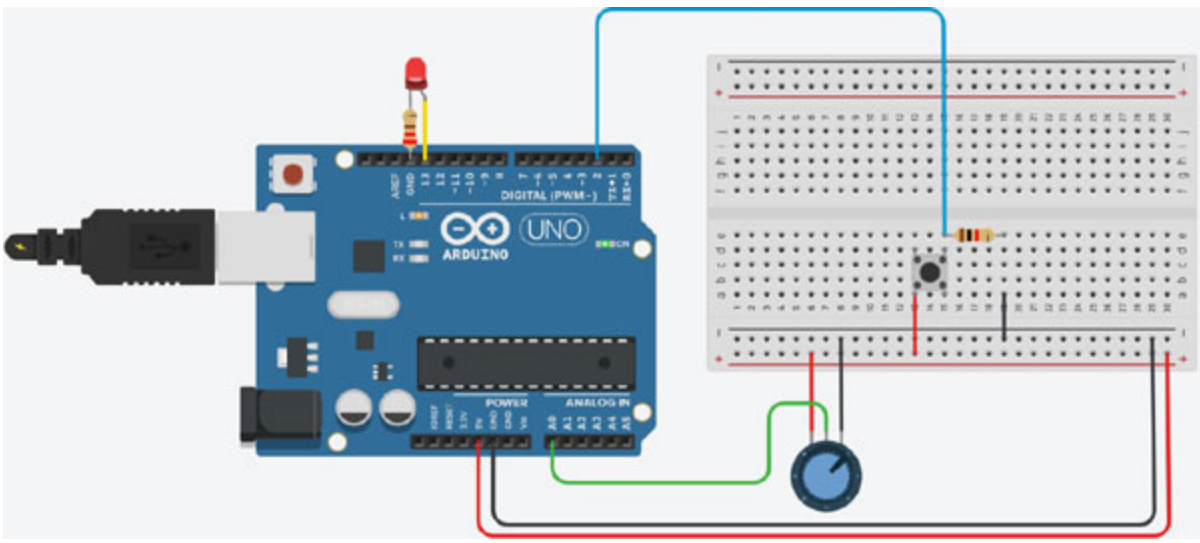


Figure 2.8: *Circuit diagram to read digital input from a pushbutton and potentiometer*

PWM pins

Pulse Width Modulation (**PWM**) is the concept of modifying the width of the output square waveform. Arduino cannot generate analog outputs, but there is a way around providing output power with different power values. There are 6 pins dedicated for such output in the Arduino UNO board. These pins are from the set of digital I/O pins. These pins have ~ sign printed for identification. PINs 3, 5, 6, 9, 10 and 11 are PWM pins.

Consider the waveforms shown in *Figure 2.9*. In this figure, we have 3 waveforms, each of which has the same time period. Any square wave has two logic levels: logic high and low. Whenever the value is high, the signal passes voltage to the output; it provides zero power. Therefore, in a given time period, if the waveform is never high (waveform 1), then this waveform will provide zero energy at the output terminal. Refer to the following figure:

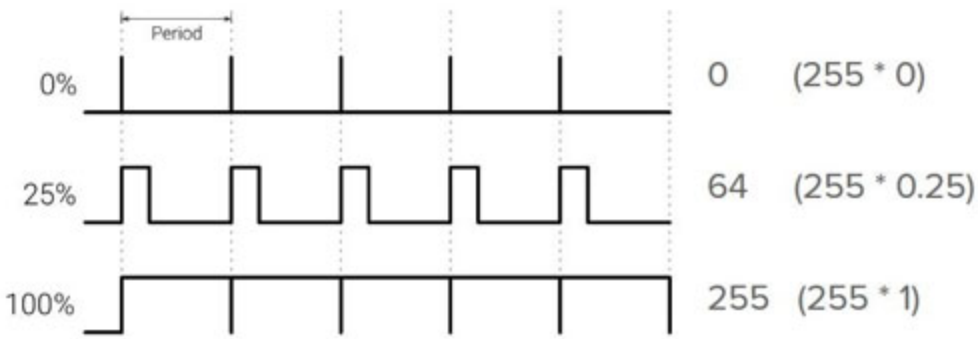


Figure 2.9: *The concept of generating different widths for a square wave*

Similarly, if the waveform is always high (waveform 3), then the waveform will provide the maximum energy. Now, using the concept of Pulse Width Modulation (PWM), we can decide the percentage of time for which the waveform will be high. The waveform will transfer the proportional energy to the output.

In Arduino UNO, we generate an output waveform of frequency 500Hz at PWM pins. We use the **`analogWrite()`** function to provide the PWM outputs at PWM pins. The function's argument is an integer in the range 0 to 255 (because it is an 8-bit value). Some of the examples to create different PWM values are as follows:

```
analogWrite(0); // implies 0% duty cycle and 0 power transfer.
analogWrite(63); // implies 25% duty cycle and 25 power
transfer.
analogWrite(127); // implies 50% duty cycle and 50 power
transfer.
analogWrite(191); // implies 75% duty cycle and 100 power
transfer.
```

```
analogWrite(255); // implies 100% duty cycle and 100 power
transfer.
```

Serial communication pins

We have discussed that when we connect an Arduino UNO with a USB cable to the computer, then along with the power, serial communication is also provided. However, when batteries power the Arduino, how do we establish the serial communication? Also, if we have to set up a serial communication with anything other than a computer, then what should be the process? In serial communication, we transmit or receive the data serially. That means we send/receive one bit at a time. Thus, two pins are enough for serial communication. In Arduino, digital pins 0 and 1 are reserved for serial communications. These pins are known as **Universal Asynchronous Receiver Transmitter** (**UART**) and **Universal Synchronous Asynchronous Receiver Transmitter** (**USART**).

We have already discussed in the recipe for a serial monitor that we need to first start the Serial communication by setting up the speed of communication in the **`setup()`** function. Some of the commonly used functions for serial communications are listed in *Table 2.3*:

Function	Brief description
`Serial.read()`	Reads the data from other devices to Arduino board
`Serial.print()`	Prints data from the Arduino sketch to the device
`Serial.println()`	Prints data and adds a new line
`if(Serial)`	Indicates whether a particular serial port is ready
`Serial.available()`	Gets the number of bytes (characters) available for reading from the serial port
`Serial.find()`	Keeps on reading the data until the target is found

Table 2.3: *Common functions used for serial communication in Arduino UNO*

Let us discuss a simple sketch demonstrating the serial communication on an Arduino UNO board. First, we will ask the user to send the data to the board, and then we will print the data back on the serial monitor. The sketch is given as follows:

```
void setup() {
 Serial.begin(9600);
}

void loop() {
 if(Serial.available()) {
  Serial.println(“ “);
  Serial.print(“I have received the message: “);
  Serial.write(Serial.read());
 }
}
```

The output of this sketch is shown in *Figure 2.10*. When the sketch is executed, the if condition is waiting for the input. We can enter any input on the serial monitor and press the **send** button. In this example, we have sent a string **“hello”**. Then, we can observe that the output is printed one character at a time. Therefore, it is verified that the serial communication happened with one character at a time. Refer to the following figure:

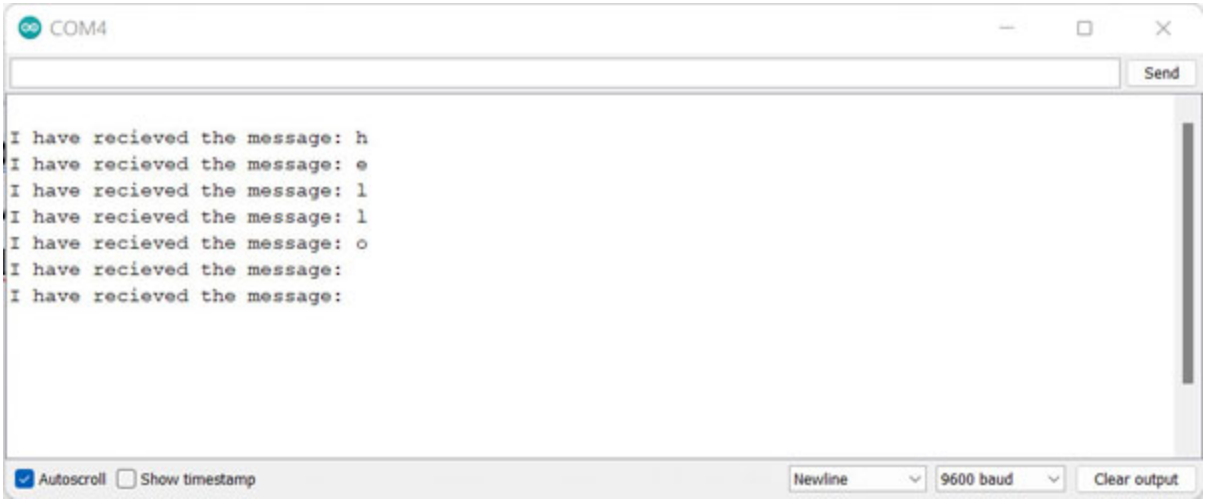


Figure 2.10: *Sending “hello” to Arduino board and then reading it back via serial communication*

There are a variety of devices that establish serial communication via the RS232 port. Such devices will require an RS232 to TTL converter.

Figure 2.11 shows an RS232 connector. This was used mainly to establish serial communication with computers. The converter is also shown in the same figure. There are 4 output pins. One for transmitting, one for receiving

and a pair of pins for ground and VCC connections. Refer to the following figure:

Figure 2.11: *An RS232 connector and RS232 to TTL converter to connect it to an Arduino board*

SPI and I2C pins

Serial Peripheral Interface (**SPI**) connects two different Arduino boards. One of the boards will control the other board. Therefore, in this configuration, the master will control the slave board. For SPI communication, we need 4 pins. On Arduino, we have 4 dedicated pins for SPI communications. Digital pins 10, 11, 12, and 13 are used for SPI communication. Let us understand their functionalities in communication. *Figure 2.12* shows the positions of SPI pins on Arduino UNO. The description of all these pins is as follows:

- Pin 10: SS (Slave Select) –the master uses this pin to enable or disable the slave
- Pin 11: MOSI (Master Out Slave In) - this pin on the master board is used to send data to the slave board.
- Pin 12: MISO (Master In Slave Out) – This pin is used by the slave to send the data to the master.
- Pin 13: SCK (Serial Clock) – this pin is used to synchronize the operations of two boards with one single clock.

Figure 2.12: *Position of SPI pins on an Arduino UNO*

Inter Integrated Circuits (**I2C**) is a similar type of communication protocol. In this communication, we use only two wires. Using this communication, we can connect any device supporting the I2C protocol to the Arduino board using a smaller number of wires. Analog pins A4 and A5 are reserved for I2C communications:

- **Pin A4: Serial Data (SDA):** This pin is used to send and receive data. Therefore, we can either send or receive the data at one time.
- **Pin A5: Serial Clock (SCL):** This pin synchronizes operations with a single clock.

The pin locations for SDA and SCL pins as follows:

Figure 2.13: *Position of I2C pins on an Arduino UNO*

Figure 2.13 shows the position of I2C pins on the board. We will discuss SPI and I2C in detail in the following chapters.

Interrupt pins

Any Arduino program keeps working until the operation is interrupted or triggered by any external event. For example, if we have executed a task and have given a delay of 20 seconds, it implies that there will not be any other action in these 20 seconds. Now, consider that you have designed a fire alarm system, and fire is detected in these 20 seconds. So, you will

expect the system to respond to this emergency and stop the current task from being executed. Interrupts provide this special feature of stopping the ongoing execution of the sketch if the trigger is detected at the interrupt pins.

On an Arduino Uno, digital pins 2 and 3 are dedicated to interrupts. *Figure 2.14* shows the positions of interrupt pins on an Arduino board:

Figure 2.14: *Position of interrupt pins on an Arduino UNO*

We use mainly two functions to work with interrupts:

- **`attachInterrupt()`**: The recommended syntax to use this function is **`attachInterrupt`**(**`digitalPinToInterrupt(pin)`**, ISR, mode).

  Here, **`digitalPinToInterrupt`** will take either 2 or 3 as the argument. We can attach our interrupt trigger to either of these pins.

  `ISR` is the function that will be called when the interrupt is detected. The name of this function can be anything.

  `Mode` is the way an interrupt is detected. The mode can be **`LOW`**, **`CHANGE`**, **`HIGH`** or **`RISING`**.

- **`detachInterrupt()`**: The function is used to turn off the given interrupt. The syntax to use the function is **`detachInterrupt(digitalPin ToInterrupt (pin))`**. Here, we will mention the PIN (2 or 3) to detach the interrupt.

Recipe 7: Convert a number into a string

Strings are a collection of characters. These characters can be anything that is typed on the screen. So, even the numbers can be considered Strings. However, by default, anything written as “1392” is considered a number

whose value is one thousand three hundred and ninety two. This is not a string with 4 characters 1, 3, 9 and 2. Occasionally, we do not want the numerical value but the character values. For example, printing the value as strings on an LCD. We will always print a 3-digit number, so we will reserve three characters for printing this output on the LCD.

Using the **`String`** data type, a number can be easily converted to a string. Consider the following code snippet:

```
String mystr1 = 2387;
```

This is the simplest declaration of a String variable called `mystr1`, giving it a value of 2387.

We can also try another method of doing the same. In this method, we will first define the number as an integer and then use Strings's **`concat()`** function to concatenate the integer value to our string:

```
int myint1 = 231;
String mystr2 = "My string is ="
mystr2.concat(myin1)
```

We have defined two separate variables in this code: **`myint1`** and **`mystr2`**. The first variable, **`myint1`**, is an integer, while **`mystr2`** is a string. However, we use the **`concat()`** function to concatenate (join) them together to generate one large string. The final value of the **`mystr2`** string will be **`"My String is = 231"`**.

Conclusion

This chapter discussed the basic concept of C programming to program an Arduino UNO board. It taught us how to create and save an Arduino sketch using Arduino IDE. We discussed different data types supported by the program and Arduino IDE. Then, we learnt how to display any serially transmitted values from Arduino on the Serial monitor of IDE. This process will always be useful in debugging the codes. We also learnt how to convert one data type into another through different recipes. Further on. We discussed the role and usage of different types of pins available on the Arduino board. This chapter helped us develop a good command over basic programming skills.

In the next chapter, we will learn about the advanced programming constructs. We will discuss the usage of conditional statements and loops. These two are one of the most vital components of programming in any language. Almost every sketch where we must take some decisions will require one or both constructs. We will also make a few projects toward the end of the chapter by combining all programming concepts.

Points to remember

C is a very versatile and popular language. We should take care of the following points while programming and Arduino IDE using C language:

- C is a case-sensitive language, so variable names should not be confused in upper and lower case
- Do not forget to add libraries in the sketch in the beginning.
- Always mention the **`setup()`** and **`loop()`** functions in every sketch.
- End statements end with a semicolon.
- Select the correct type of datatype for variables. For example, we cannot use an integer to store characters.
- The selection of data type should be based on the type of inputs. For example, digital input/outputs cannot be assigned as integers or strings.
- While uploading the code, ensure that the board and ports are selected accurately.

Multiple choice questions

1. **In Arduino IDE, IDE stands for which of the following?**
    a. Integrated Digital Environment
    b. Integrated Development Environment
    c. Instruction Development Environment
    d. Integrated Development Embedded
2. **An Arduino sketch does not have the traditional `main()` function that's generally present in C programs.**

   a. True
   b. False

3. **Arduino IDE consists of two functions. What are they?**
   a. Build() and loop()
   b. Setup() and build()
   c. Setup() and loop()
   d. Loop() and build() and setup()D

4. **`analogWrite()` is used to do which of the following?**
   a. Generate analog voltage at analog output
   b. Write input analog voltage in memory
   c. Generate square wave
   d. Generate PWM

5. **Which of the following functions is used to set any pin in the state of HIGH/LOW ?**
   a. digitalRead
   b. analogWrite
   c. pinMode
   d. digitalWrite

Answers

1. **b**
2. **a**
3. **c**
4. **d**
5. **d**

CHAPTER 3

Advanced Programming Construct

Introduction

We have a good command of C programming concepts now. The programming knowledge gained in the previous chapter has helped us build simple applications using Arduino. Now, we will learn the complex programming concepts and recipes based on those concepts.

This chapter will have a new set of recipes for advanced programming constructs. We will now discuss the use of conditional statements and loops. We can write most Arduino sketches using these two vital programming constructs. A few of the recipes were partially explained in the previous chapter due to a lack of knowledge of these two constructs. Now, we will have full-fledged recipes for those concepts as well. We will discuss the process of interrupt handling and reading analog values in detail. We will make circuits with Pulse Width Modulation (PWM) and sound generation. So, let us get started with these advanced recipes.

Structure

In this chapter, we will discuss the following recipes:

- Performing different actions based on two different variable values
- Performing multiple actions based on multiple values of a variable
- Repeating an action, a fixed number of times
- Repeating an action till another event occurs
- Grouping and splitting different variables
- Splitting text from a group
- Interrupting an ongoing event with another event
- Generating different tones using Arduino Uno
- Changing the intensity of an LED

- Sending binary values through serial communication
- Taking multiple decisions using case statements
- Generate the knight rider sequence using 7 LEDs
- Controlling five different LEDs using serial communication through PC

Objectives

By the end of this chapter, the learner will be able to do the following:

- Use conditional statements to take multiple decisions
- Use case statements to take different actions based on the value of a variable
- Utilize loops to perform repetitive tasks
- Know the difference between different kind of loops
- Know about PWM signal generation and its applications

Recipe 1: Performing different actions based on two different variable values

In Arduino, we use different variables and their values to decide certain operations, for example, if we are monitoring a certain location's temperature or humidity. Based on the temperature value, we have decided on two different operating modes for the exhaust fan. Therefore, there is a condition, and we decide our actions based on the condition.

In programming, conditional statements are implemented using the **`if`** statement. Let us understand this statement with the help of the following example. In *Figure 3.1*, we have connected one potentiometer with an Arduino board, and there is an LED connected on the same board. Please refer to the following figure:

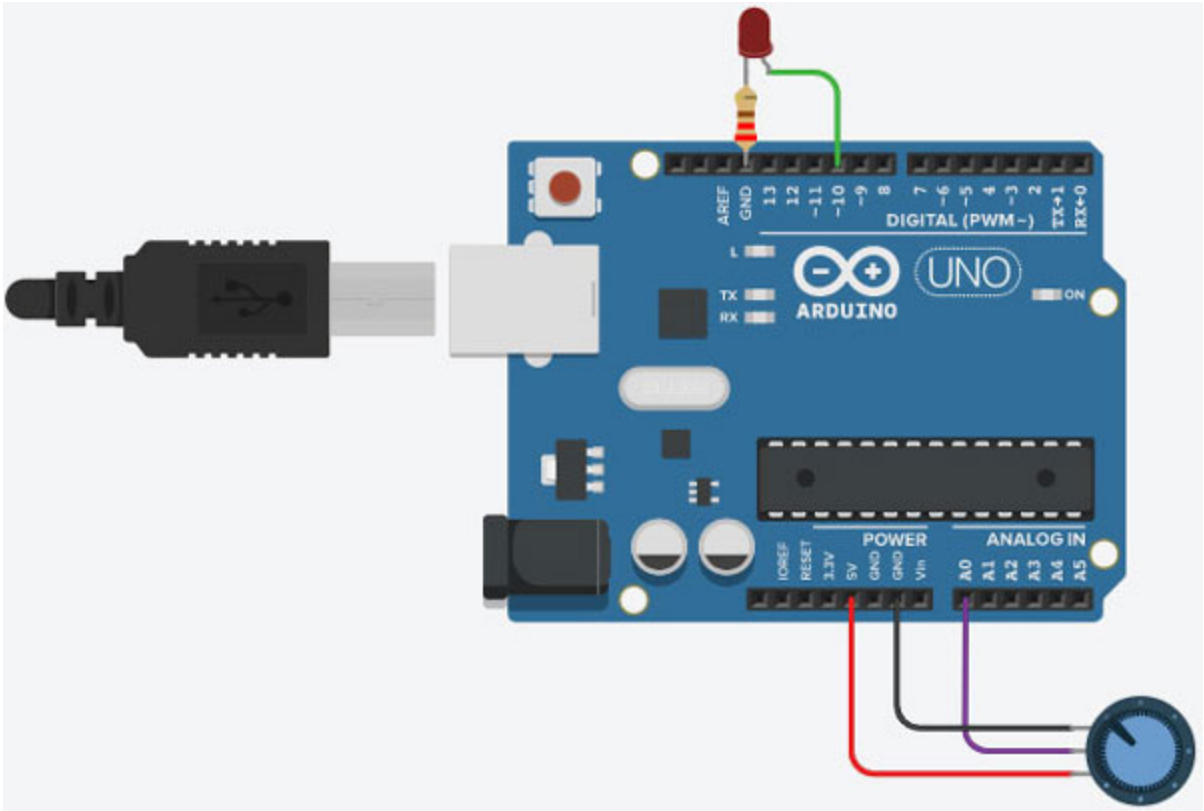


Figure 3.1: *A potentiometer-controlled LED*

A potentiometer is an electrical device that is equivalent to a series combination of two registers. The potentiometer has three terminals. One of them is connected to the ground, and the other extreme is connected to the power supply, which is 5 Volt in this case. Therefore, the voltage at the central terminal will always be between 0 to 5 Volt, as shown in *Figure 3.2*:

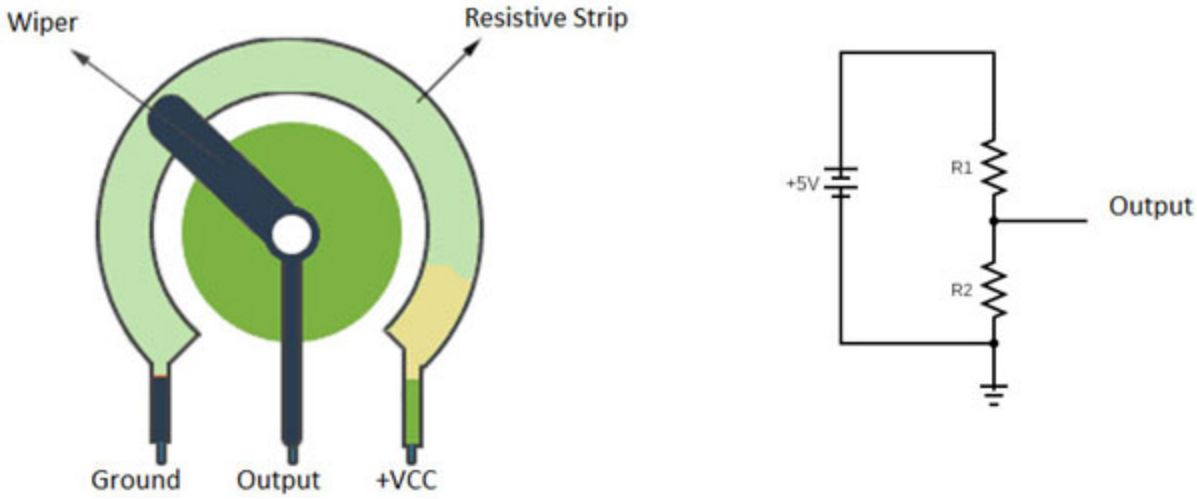


Figure 3.2: Connections and equivalent circuit of a potentiometer

The output of the potentiometer is connected to analog pin A0. Now, we will receive the input between 0 to 1023 at A0. Then, in the sketch, we use the **`if`** statement to decide the state of the LED. We can decide any threshold value and use conditional statements. We decide the operation of the LED. Let us understand the sketch now:

```
int PotVal = 0;
void setup()
{
 pinMode(A0, INPUT);
 pinMode(10, OUTPUT);
}

void loop()
{
 PotVal = analogRead(A0);
 if (PotVal < 500) {
  digitalWrite(10, HIGH);
  delay(1000); // Wait for 1000 millisecond(s)
 } else {
  digitalWrite(10, LOW);
  delay(1000); // Wait for 1000 millisecond(s)
 }
}
```

Here, we have declared an integer variable **`PotVal`**. This will read the analog input at pin **`A0`** from the potentiometer. In the **`loop()`** function, we have used the conditional statement. We have taken a simple decision that if the value of **`PotVal`** is less than 500, the LED connected at digital pin 10 will glow; otherwise, it will not.

Recipe 2: Performing multiple actions based on multiple values of a variable

In the previous recipe, we made two decisions (turning on or off the LED) based on two sets of values of the input received at the analog pin. We can

also make multiple decisions using conditional statements. In this process, we use the **`if-elseif-else`** statement. Let us control two different LEDs using these statements now. The connections are shown in *Figure 3.3*:

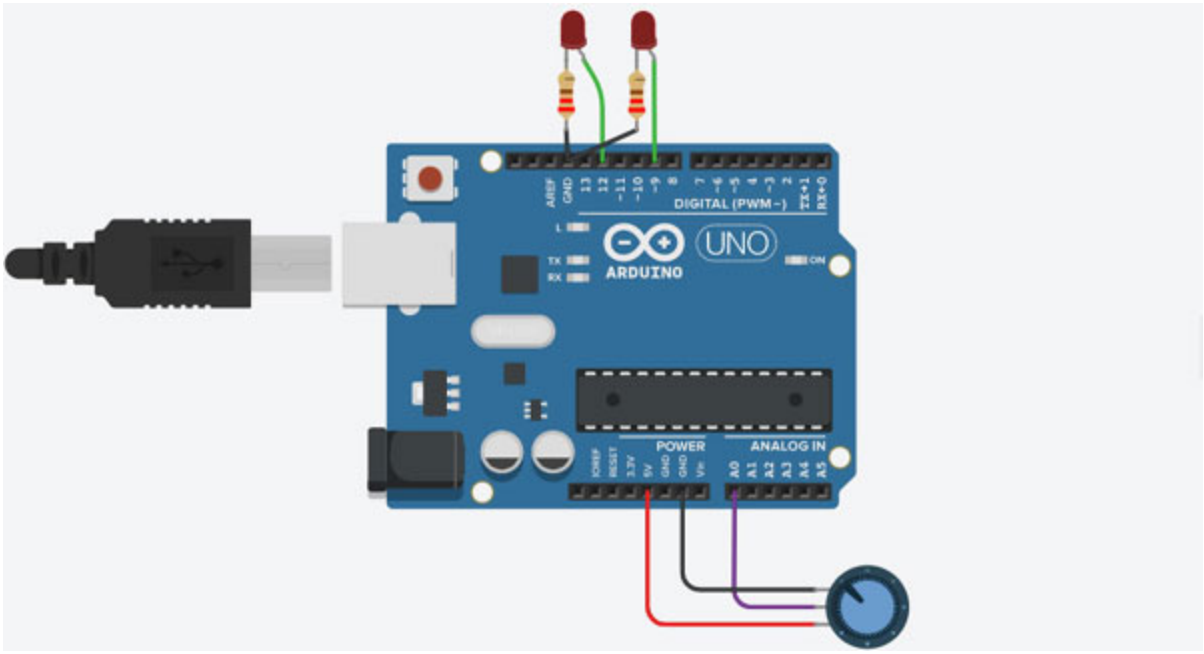


Figure 3.3: *Performing multiple operations based on multiple values of a variable*

We have used a very similar setup in this recipe as well. We have connected two different LEDs at digital pins 9 and 12. The potentiometer provides the input at analog pin A0. The sketch is as follows:

```
int PotVal = 0;
void setup()
{
 pinMode(A0, INPUT);
 pinMode(9, OUTPUT);
 pinMode(12, OUTPUT);
}

void loop()
{
 PotVal = analogRead(A0);
 if (PotVal > 800) {
  digitalWrite(9, LOW);
```

```
    digitalWrite(12, LOW);
    delay(1000); // Wait for 1000 millisecond(s)
  }
  else if ( 600 < PotVal && PotVal < 800){
    digitalWrite(9, HIGH);
    digitalWrite(12, LOW);
    delay(1000); // Wait for 1000 millisecond(s)
  }
  else if ( 400 < PotVal && PotVal < 600){
    digitalWrite(9, LOW);
    digitalWrite(12, HIGH);
    delay(1000); // Wait for 1000 millisecond(s)
  }
  else {
    digitalWrite(9, HIGH);
    digitalWrite(12, HIGH);
    delay(1000); // Wait for 1000 millisecond(s)
  }
}
```

We can observe in this sketch that we are performing four different operations based on different values of **`PotVal`**. The **`else if`** statement provides the flexibility of making multiple decisions based on the multiple values of a variable.

Recipe 3: Repeating an action a fixed number of times

In programming, we use loops to repeat certain statements. We use a **`for`** loop to repeat statements a fixed number of times. To demonstrate the functionality, we will print "**`Hello World`**" a fixed number of times and then print "*Well done*" at the end:

```
void setup ( )
{
  int i;
  Serial.begin(9600);
  for (i = 0; i < 5; i = i + 1)
```

```
  {
   Serial.println("Hello World");
  }
  Serial.println("Well Done");
}
void loop ( )
{
}
```

The output of the sketch is as follows:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Well Done
```

We can observe that we have not mentioned anything in the **`loop()`** functions, so the events are not repeated.

Recipe 4: Repeating an action till another event occurs

Repeating an action a fixed number of times is a predetermined task. We know that we must only perform this action several times. However, there are occasions when we are uncertain about the number of times we must repeat a certain task. For example, we must keep the water pump on until the water level in the storage tank reaches a certain level. Or, we must keep the air conditioner running until the temperature drops below 26 degrees. The **`For`** loop is not a useful tool in such situations; we should use a **`while`** loop. The setup and loop functions describes these steps:

```
void setup()
{
 Serial.begin(9600);
}
int myVar = 0;
void loop(){
 while (myVar < 100) {
```

```
    Serial.print("The value of myVar = ");
    Serial.println(myVar);
    myVar = myVar+15;
  }
}
```

The output is as follows:

```
The value of myVar = 0
The value of myVar = 15
The value of myVar = 30
The value of myVar = 45
The value of myVar = 60
The value of myVar = 75
The value of myVar = 90
```

We can observe that the value of **`myVar`** is printed until it is less than 100. The process is stopped after that. A **`while`** loop is quite useful whenever we must make decisions based on the sensor values. In the incoming chapters, we will deal with recipes involving multiple sensors. Loops and conditional statements are vital for such recipes.

Recipe 5: Grouping and splitting different variables

In this recipe, we will learn how to reduce the number of lines in an Arduino sketch to access multiple variables. Consider the situation where we must control six different LEDs using six different buttons. We need six pins on the Arduino board, which will take inputs from the buttons and six more pins to generate outputs for the LEDs. We need to define the **`pinMode`** and the related statements separately. This will make the code lengthy. The code is not complex, but it will be lengthy. To improve it, we can group the input pins into an array and output pins into another array. This grouping is done using for loop.

The circuit diagram of the recipe is shown in *Figure 3.4*:

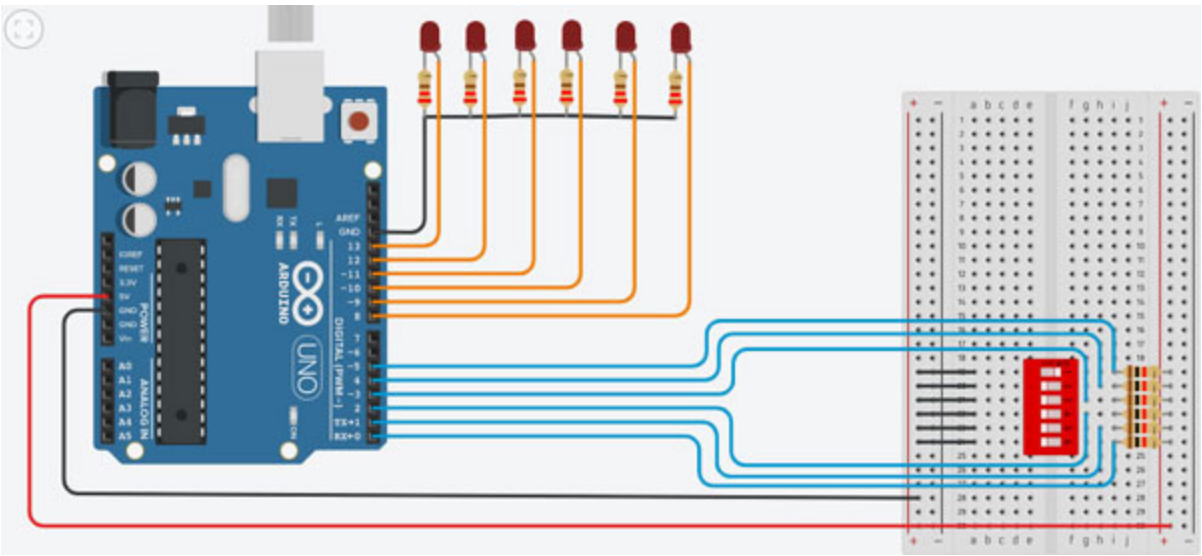


Figure 3.4: *Controlling six LEDs with six switches*

The sketch for the recipe is given below. We have described 6 input pins and 6 output pins. These pins are connected to the respective switches and LEDs:

```
int inPins[] = {0,1,2,3,4,5};
int outPins[] = {8,9,10,11,12,13};
void setup()
{
 for(int i = 0; i < 6; i++)
 {
  pinMode(inPins[i], INPUT);
  pinMode(outPins[i], OUTPUT);
 }
}
void loop()
{
 for(int i = 0; i<6; i++)
 {
  int in_value = digitalRead(inPins[i]);
  digitalWrite(outPins[i],in_value);
 delay(100); // Wait for 1000 millisecond(s)
 }
```

```
}
```

In this recipe, we created an array **inPins**, which grouped all six input pins, and another array **outPins** grouped six output pins. Now, we have used index values in the array to read the values and write the values on output pins.

Recipe 6: Splitting text from a group

In the previous recipe, we grouped different integers into an array. In this recipe, we will split the text of a String by identifying the separating symbols. For example, in a large paragraph, sentences are made by identifying periods, question marks, exclamation marks and similar symbols. Here, we will split the text based on the '*!*' Symbol detected in the String. The code foe the recipe is as shows:

```
String str1 = "Oh My God! Yess! Thats right! Bravo";
String temp = str1;
int Pos_symbol =0; // to store the position of !
void setup()
{
 Serial.begin(9600);
 Serial.print("The original String = ");
 Serial.println(temp);
 do
 {
  Pos_symbol = str1.indexOf('!');
  if(Pos_symbol != -1)
  {
   Serial.println( str1.substring(0,Pos_symbol));
   str1 = str1.substring(Pos_symbol+1, str1.length());
  }
  else
  {
   if(str1.length() > 0)
    Serial.println(str1); // if there is text after the last
    comma,
```

```
    }
    delay(1000);
  }
  while(Pos_symbol >=0);
}
void loop()
{}
```

The output on the serial monitor is as follows:

```
The original String = Oh My God! Yess! Thats right! Bravo
Oh My God
  Yess
  Thats right
  Bravo
```

Recipe 7: Interrupting an ongoing event with another event

Interrupts are an integral part of programming an Arduino. Whenever an interrupt is detected, all other executions are stopped, and the compiler shifts to compiling the function called by the interrupt function. The function to detect interrupts is **`attachInterrupt()`**. It takes three arguments: PIN of interrupt, function to be called and mode of interrupt detection. We have attached a push button in this recipe at the interrupt pin (digital pin 3). The sketch simply prints integers from 0 to 19 in a forever loop. Whenever the rising edge is detected at the interrupt pin, the execution will shift to the **`fun1`** function. In the **`fun1`** function, we print "**`Interrupt detected`**", turn on the LED at pin 13 and give a delay of 2 seconds. After these steps, the execution will continue from the same count.

The circuit diagram is shown in *Figure 3.5*. The sketch is given after that:

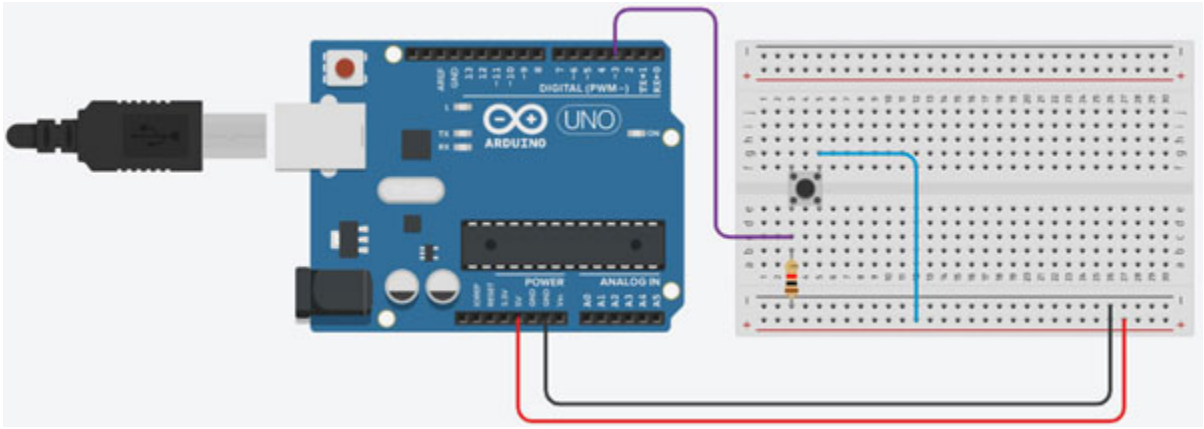


Figure 3.5: *Connections for providing an interrupt to Arduino using a push button*

The coding for interrupt-based sketch is shown below. We have taken the interrupt at digital pin 3. The interrupt is detected with RISING edge and then the function **`fun1`** is called. The sketch is as follows:

```
void setup() {
  pinMode(13, OUTPUT);
  pinMode(3, INPUT);
  Serial.begin(9600);
  attachInterrupt(digitalPinToInterrupt(3), fun1, RISING);
}
void loop() {
  digitalWrite(13,LOW);
  for(int i = 0; i < 20; i++)
  {
    Serial.println(i);
    delay(1000);
  }
}
```

The output is as follows:

```
0
1
2
3
Interrupt detected
4
```

```
5
6
Interrupt detected
7
8
Interrupt detected
9
10
11
12
13
Interrupt detected
14
15
16
17
Interrupt detected
18
19
0
1
2
```

In this recipe, we used the rising edge of the signal at the interrupt pin to detect the interrupt. The interrupt can also be detected at the LOW value, CHANGE in the value, or at the FALLING edge.

Recipe 8: Generating different tones using Arduino uno

A violin string moves back and forth rapidly, moving the surrounding air and generating a tone. The musical note A is heard when a violin string moves 440 times per second in the violin. The number of vibrations per second is called frequency. These signals are converted into sound with the help of speakers.

A "*pure*" tone is a sine wave. However, an Arduino is a digital device; it can generate square waves only. The resulting voltage is then a square

wave, high for one-half the period and low for one-half the period. The duration of one cycle is called the period. Therefore, we use one of the PWM pins to generate a tone. We use the **`tone()`** function to generate the tone. The syntax is as follows:

```
tone( pin, frequency, duration).
```

So, this will send a pulse with the specified frequency to the specified pin for the given duration. The circuit diagram is shown in *Figure 3.6*, and the sketch follows it:

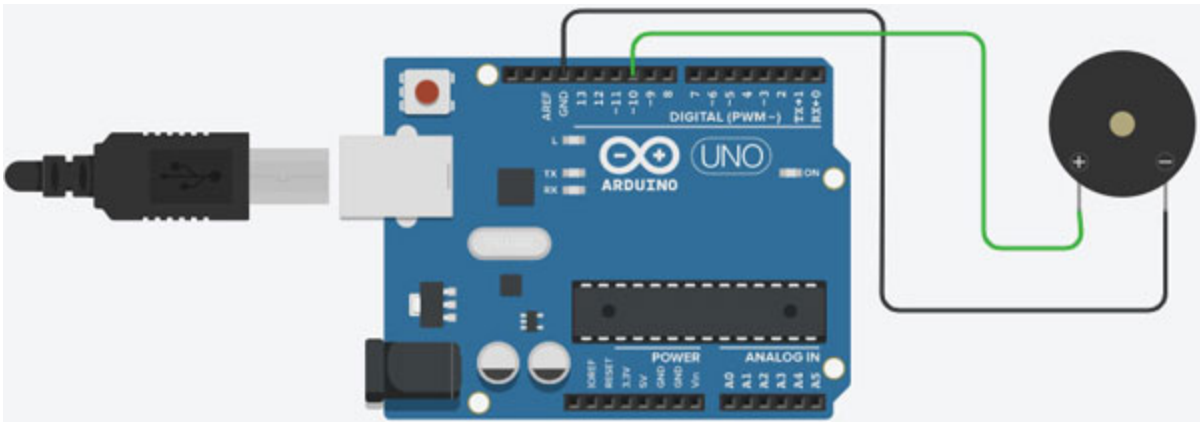


Figure 3.6: *Tone generation using an Arduino board*

The sketch to generate a specific tone using an Arduino is as follows:

```
void setup()
{
  pinMode(10, OUTPUT);
}
void loop()
{
  tone(10, 500, 1000);
  delay(1000);
  tone(10, 1500, 1000);
  delay(1000);
}
```

The sketch will produce a tone of 500 Hz for one second and 1500 Hz for the next second. The process is repeated forever in this sketch.

Recipe 9: Changing the intensity of an LED

Generally, we connect the LED to a digital pin to switch it on or off. However, if we want to switch on an LED with a particular intensity, we must use a PWM pin and provide a PWM value. As discussed earlier, we can provide 0 to 255 as the output at the PWM pin. The intensity of the LED will depend on the PWM value. The LED will not glow at 0, and it will glow with maximum intensity at 255. A sample sketch to slowly increase the intensity of the LED is given as follows. Once the intensity reaches the maximum value, we set it back to 0. The operation keeps repeating.

In this sketch, we have used PIN 6 for PWM output. The intensity of LED is slowly increased (with a delay of 10ms):

```
void setup()
{
 pinMode(6, OUTPUT);
}
void loop()
{
 for(int i = 0; i<256; i++)
 {
  analogWrite(6,i);
  delay(10);
 }
 analogWrite(6,0);
 delay(1000);
}
```

Recipe 10: Sending binary values through serial communication

When we send the messages through serial communication, we send one byte at a time. That means if we must send a message larger than 8-bits, we will need more than one cycle to send the data. In this recipe, we will send a 10-bit value, typically received at the input to analog pins, via serial

communication. In this recipe, we use the **`Serial.write()`** function. The function will send only 8-bit messages at a time. So, first, we need to split our 10-bit message into two bits.

A 10-bit message can be split into two bytes using the modulus operator. Then, we will send one byte at a time. The sketch is given as follows:

```
void setup() {
  Serial.begin(9600);
}
void loop() {
int inpVal = analogRead(A2);
byte Byte1 = inpVal/256; // Remainder of the division
byte Byte2 = inpVal % 256; // Quotient of the division
Serial.write(Byte1);
Serial.write(Byte2);
}
```

In this recipe, initially, we took the 10-input from A2 and stored it into an integer variable **`inpVal`**. Now, the integer is divided by 256 (2^8). The remained will always be less than 256 and will provide the least significant 8 bits. These bits are stored in a new byte type variable, **`Byte1`**. The higher 2 bits are stored in **`Byte2`**.

For example, if **`inpVal`** = 1000, which is “11,1110,1000”, the last 8 bits, “1110,1000”, will be stored in **`Byte1`**, and **`Byte2`** will store the value “0000,0011”.

Recipe 11: Taking multiple decisions using case statements

Another useful conditional statement in Arduino programming is a **`case`** statement. In Arduino, a case statement is combined with the **`switch`** statement. In this recipe, we have used the combination of **`switch`** and **`case`** to decide the number of times an LED will blink. The inputs are taken from the keyboard and stored in **`inCode`**. The **`switch`** statement will switch to different cases based on the value of **`inCode`**. We have mentioned three different cases. In two cases, we are switching to different functions as well.

It must be noted that there must be a **`default`** case in the sketch. It will be used to handle any case that is not defined. The sketch for the recipe is as follows:

```
void setup()
{
 Serial.begin(9600);
 pinMode(9, OUTPUT);
}
void loop()
{
 if ( Serial.available())
 {
  int inCode = Serial.read();
  switch(inCode)
  {
   case ‘0’:
   digitalWrite(9,LOW);
   Serial.println(“LED is switched off”);
   break;
   case ‘1’:
   Serial.println(“LED will blink once”);
   blink1();
   break;
   case ‘2’:
   blink2();
   Serial.println(“LED will blink twice”);
   break;
   default :
   Serial.println(“Please enter only 0, 1, or 2”);
   break;
  }
 }
}
void blink1()
```

```
{
 digitalWrite(9,HIGH);
 delay(1000);
 digitalWrite(9,LOW);
 delay(1000);
}
void blink2()
{
 digitalWrite(9,HIGH);
 delay(1000);
 digitalWrite(9,LOW);
 delay(1000);
 digitalWrite(9,HIGH);
 delay(1000);
 digitalWrite(9,LOW);
 delay(1000);
}
```

The output with input sequences 1, 5, 2, 2, and 0 is as follows:

```
LED will blink once
Please enter only 0, 1, or 2
LED will blink twice
LED will blink twice
LED is switched off
```

Recipe 12: Generate the knight rider sequence using 7 LEDs

A knight rider circuit is a fancy glowing pattern of LEDs. In this circuit, we take 7 LEDs to arrange in a line. Initially, all the LEDs are off. Then, one of the corners LED is switched on. In the next iteration, the second LED is also switched on. Then, the third is switched on, and we continue switching the LEDs on until we reach the last one. Once all the LEDs are on, we start switching them off in the backward direction.

Figure 3.7 shows the circuit diagram for the 7-LEDs-based knight rider circuit. The LEDs are attached to the digital pins 6 to 13. We can even use all 14 pins to make a larger knight rider circuit. Refer to the following figure:

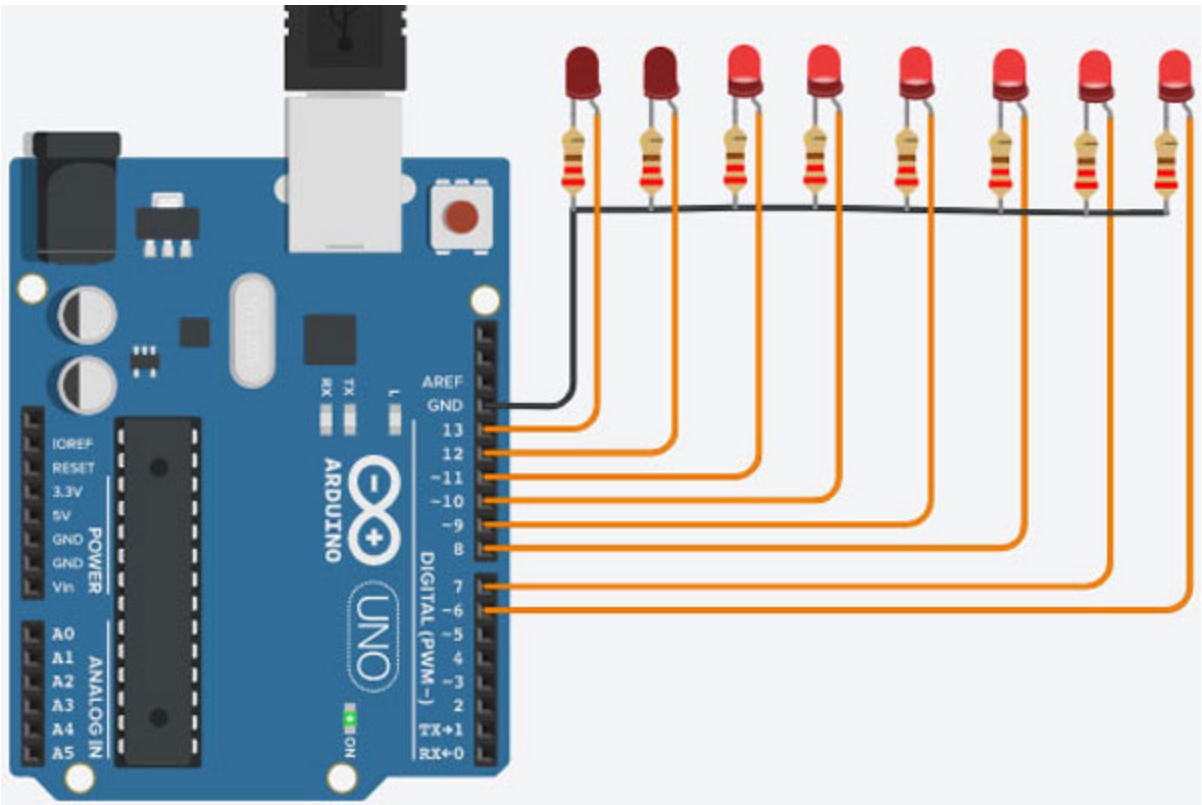


Figure 3.7: *A knight ride circuit with 7 LEDs*

The sketch for the recipe is given as follows. In this sketch, we have used **`for`** loop to declare PIN 6 to 13 as output pins in the **`setup()`** function. In the **`loop()`** function, we have used two **`for`** loops. One is used for incremental glowing LEDs, and another is for decremental switching off.

```
void setup()
{
  for(int i = 6; i <= 13; i++)
  {
    pinMode(i, OUTPUT);
  }
}
```

```
void loop()
{
 for(int i = 6; i <= 13; i++)
 {
  digitalWrite(i,HIGH);
  delay(100); // Wait for 1000 millisecond(s)
 }
 for(int i = 13; i >=6; i--)
 {
  digitalWrite(i,LOW);
  delay(100); // Wait for 1000 millisecond(s)
 }
}
```

Recipe 13: Controlling 5 different LEDs using serial communication through PC

In this recipe, we will control the operation of 5 different LEDs using the computer's keyboard. We will assign numerical keys 1, 2, 3, 4 and 5 to control the operations of 5 different LEDs numbered 1 to 5. The same key will be used to turn the assigned LED on and off. We will introduce the concept of state to control both operations (switching on and off) using a single key. We will track if the LEDs are currently switched on or off and store the state value. Whenever the key is pressed again, the state will be changed to the opposite value.

Figure 3.8 shows the circuit connections for the recipe:

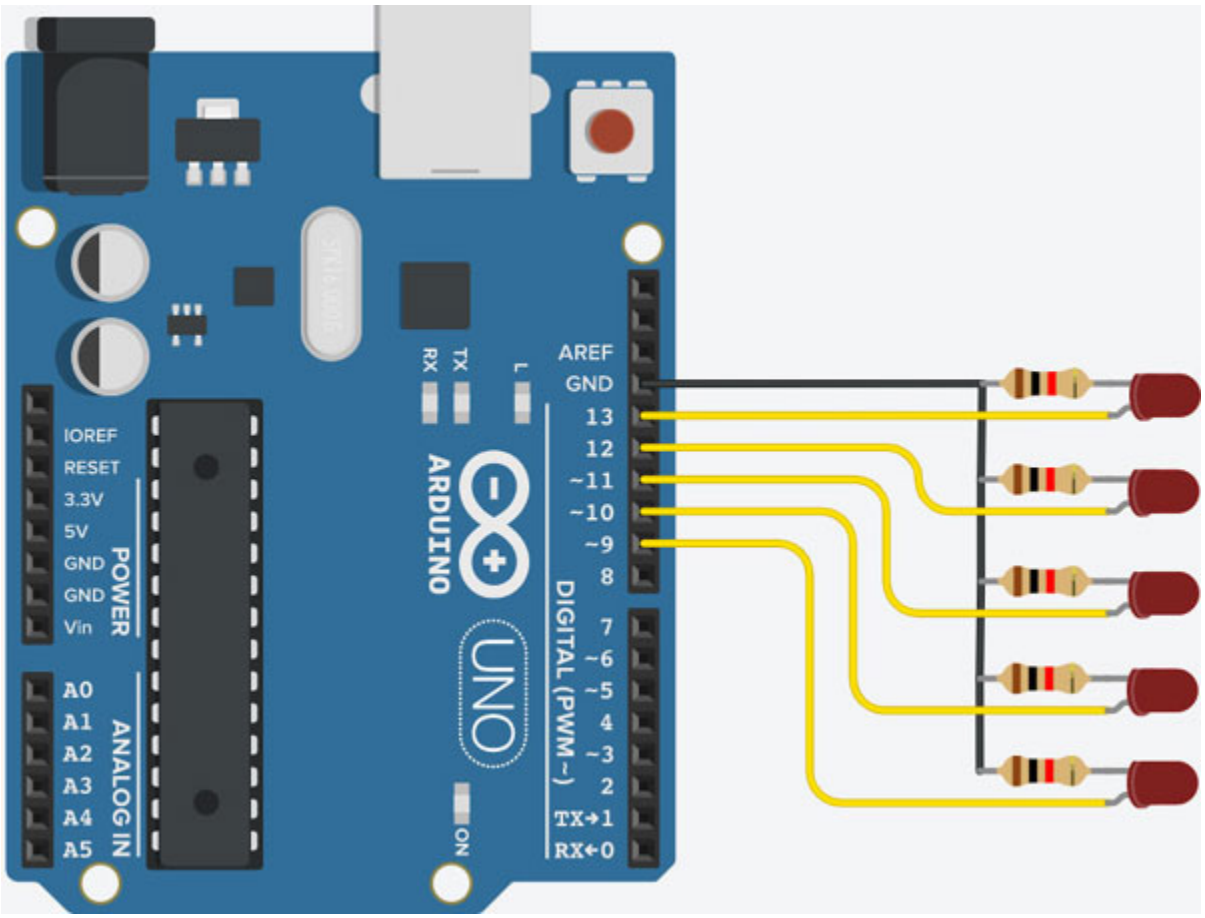


Figure 3.8: *Circuit diagram to control 5 LEDs using 5 different keyboard values*

The sketch for the recipe is given as follows. We define an integer **`in1`** to store the value of serial inputs. We have selected numeric values 1, 2, 3, 4 and 5 from the keypad as our control pins. When we check their serial values with the help of **`serial.read()`** function, the values are 49, 50, 51, 52 and 53, respectively. Therefore, the if statement will be controlled by these serial values. We use **`state1`**, **`state2`**, **`state3`**, **`state4`** and **`state5`** to store the current state of each LED and flip them whenever the respective button is pressed. The complete sketch for the recipe is as follows:

```
int in1 = 0;
bool state1 = LOW, state2 = LOW, state3 = LOW, state4 = LOW,
state5 = LOW;
void setup(){
  pinMode(13, OUTPUT);
  pinMode(12, OUTPUT);
```

```
  pinMode(11, OUTPUT);
  pinMode(10, OUTPUT);
  pinMode(9, OUTPUT);
  Serial.begin(9600);
}
void loop(){
  if (Serial.available()>0) {
    in1 = Serial.read();
    if (in1 == 49){
      state1= !state1;
      digitalWrite(13, state1);
      delay(500);
    }
    else if(in1 == 50){
      state2= !state2;
      digitalWrite(12, state2);
      delay(500);
    }
    else if(in1 == 51){
      state3= !state3;
      digitalWrite(11, state3);
      delay(500);
    }
    else if(in1 == 52){
      state4= !state4;
      digitalWrite(10, state4);
      delay(500);
    }
    else if(in1 == 53){
      state5= !state5;
      digitalWrite(9, state5);
      delay(500);
    }
    else {
```

```
    digitalWrite(13, state1);
    digitalWrite(12, state2);
    digitalWrite(11, state3);
    digitalWrite(10, state4);
    digitalWrite(9, state5);
    delay(500);
   }
  }
 }
```

Conclusion

This chapter concludes the programming concepts section of this book. This chapter taught us complex programming concepts with the help of interesting recipes. We learned the conditional statements and used them to perform different actions based on a variable's value. Then, we discussed various loops in programming. The loops helped us in performing repetitive tasks effectively. We also learned the usage of interrupts to overpower the ongoing process with high-priority interrupts. Additionally, we discussed pulse width modulation concepts. Towards the end, we developed a few recipes to generate interesting patterns using different LEDs.

The next chapter will explore new hardware integration with the Arduino board. We will discuss the connection of switches ad displays with the Arduino board. Some special displays will require adding specific libraries to the sketch. We will also discuss displaying different types of data on various display units.

Points to remember

C is a very versatile and popular language. We should take care of the following things while programming and Arduino IDE using C language:

- Use proper **`if`**, **`else-if`**, and **`else`** combinations while using if statements
- Ensure that you do not write a condition that can run the loop infinite times; it will put an additional computational load on the

microcontroller

- A proper combination of loops and conditional statements should be used
- Interrupts can be used only at pins 2 and 3
- PWM works only for 6 out of 14 digital pins

Multiple choice questions

1. **delay(10000) results in a delay of _____.**

   a. 100000 seconds

   b. 100 seconds

   c. 1 second

   d. 10 seconds

2. **When writing a sketch, how do you decide which features belong in the setup function and which ones belong in the loop function?**

   a. Features that need to run continuously go in loop

   b. Features that need to be initialized go in loop

   c. Features that need to run continuously go in setup

   d. Features that need to be initialized go in setup

3. **What will the following code do?**

```
void loop()
{ Serial.begin(9600);
 String stir = “12”;
 stir +=1;
 int analogValue = 0;
 delay(1000);
 Serial.println(stir);
}
```

   a. Print 12 on the serial monitor

   b. Print 121 on the serial monitor

   c. Give syntax error

d. Print 13 on the serial monitor

4. **Which function in the Arduino IDE is used to set any pin in output or input state?**

   a. delay
   b. analogRead
   c. pinMode
   d. digitalWrite

5. What will the following code do?

```
void setup()
{
}
void loop()
{ Serial.begin(9600);
 int analogValue = 0;
 delay(1000);
 Serial.println(analogValue);
}
```

   a. Print 0 after every second on the serial monitor
   b. Print 0 once on the serial monitor
   c. Not print anything on the serial monitor
   d. Will not compile and will give a syntax error

Answers

1. **d**
2. **a**
3. **b**
4. **c**
5. **a**

CHAPTER 4

Switches and Displays

Introduction

In the previous section spanning two chapters, we discussed programming concepts for programming an Arduino board. These chapters helped us in programming a board with a different set of commands. Now, we will move to the next important part of Arduino-based recipes: hardware integration. This section will discuss integrating various external hardware components as input or output devices with an Arduino board. We will learn their circuit connections and coding to obtain correct inputs and deliver accurate outputs.

In this chapter, we will learn about switches, buttons and display devices. Switches are the simplest and most widely used input devices. Switches and buttons are available in various forms and shapes. We will learn the connections of witches with power supplies and how to supply input voltages from a switch to an Arduino board. We will also learn about different display devices. The digital output from an Arduino board can be easily displayed using **Light Emitting Diode** (**LED**). We can use multiple LEDs or even a seven-segment display for multiple outputs. We will look at how to use a **Liquid Crystal Display** (**LCD**) to display characters and numbers. In the end, we will learn to use organic **Light-Emitting Diodes** (**OLED**) and fancy display units, like NeoPixels.

Structure

This chapter will have the following recipes related to switches and display units:

- Reading different voltage values from a switch
- Calculating the time for which a pushbutton is pressed
- Generating red, green, and blue colors using an RGB LED

- Generating millions of random colors using RGB LED
- Driving a 7-segment LED display
- Displaying a blinking message on a 16x2 LCD
- Displaying a left scrolling message on a 16x2 LCD
- Displaying a rotating sequence using a NeoPixel LED
- Generating different voltage values using a potentiometer and displaying them on an LCD
- Designing a 6-level water level indicator using an LED array

Objectives

By the end of this chapter, the learner will be able to do the following:

- Identify different switches and buttons and use them in circuits.
- Control display devices using switches or buttons.
- Use different display devices like 7-segment display, LCD, and OLED.
- Handle fancy display devices like NeoPixels.
- Integrate switches and LEDs to develop a water level detector system.

Recipe 1: Reading different voltage values from a switch

Switches are present in almost every room. We use them to control electrical appliances. Switches are present in vehicles and even game controllers. The primary purpose of a switch is to establish or stop the current flow between two points. A slider switch is one of the simplest switches used in electrical and electronics circuits. As shown in *Figure 4.1*, this witch has three terminals. The central terminal is used as the output terminal. The corner terminals are used to connect to two different power supplies (voltages). In the digital world, we connect one terminal to the ground and one to VCC. The connection shows that if we slide the sliding contact to one side, then two terminals get connected. Therefore, the central terminal will get the voltage of the connected terminals. Similarly, if we slide that sliding contact to the opposite side, the output terminal will get the voltage of the other corner terminal. Refer to the following figure:

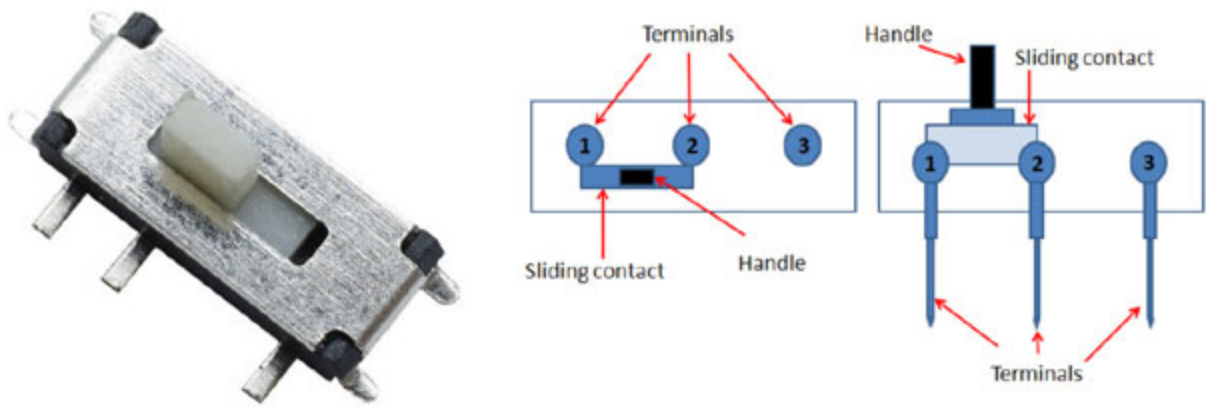


Figure 4.1: *A sliding switch and its internal connections*

Now, let us design a simple circuit where we will connect 5 volts and ground to these 2 corner terminals of a switch. We will read the output voltage using Arduino and display it on the serial monitor. The circuit diagram for this recipe is shown in *Figure 4.2*:

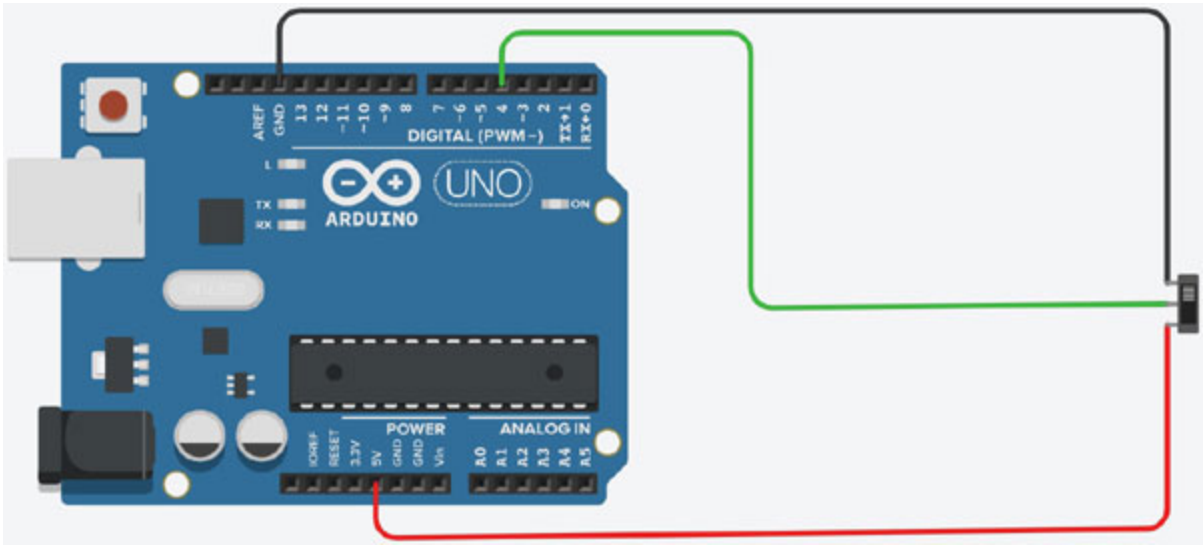


Figure 4.2: *The connections of the sliding switch with an Arduino board*

In *Figure 4.2*, one corner terminal of the switch is connected to 5V, and the other corner is connected to the ground. The central terminal is connected to digital pin 4. The sketch for the recipe is given as follows. In this sketch, we have declared a variable **`Switch_val`**. It is reading the digital input coming at pin 4. We simply print the value received at pin 4 using the serial monitor.

```
int Switch_val = 0;
```

```
void setup()
{
 pinMode(4, INPUT);
 Serial.begin(9600);
}

void loop()
{
 Switch_val = digitalRead(4);
 Serial.print("The voltage coming from the sliding switch is
 ");
 Serial.println(Switch_val);
 delay(1000);
}
```

Recipe 2: Calculating the time for which a pushbutton is pressed

A pushbutton is a variation of a switch. The pushbutton will establish the connection between two sides only when it is kept pressed. As soon as we release the button, the connection between the two sides is broken. Refer to the following figure:

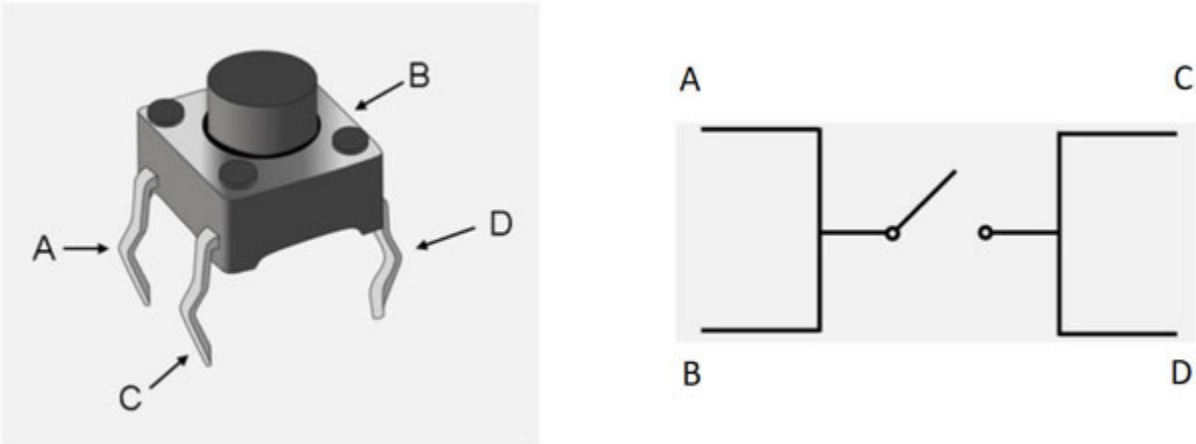


Figure 4.3: *A push button and its internal circuit connections*

Figure 4.3 shows the structure and internal circuitry of a push button. There are 4 physical pins on 4 corners of the square-shaped body. Pins A and B

are always connected; pins C and D are always connected. The connection between the pins is established when we push the button. We can use this button to pass the value of the voltage signal from one side to another by pressing it.

In this recipe, we will measure the time for which the pushbutton is pressed. We will connect the output of the pushbutton to one of the digital pins. By default, the output from the pushbutton will be 0V. Whenever we press the button, the output will be 5V. Therefore, we will start a timer and counter to count the time for which the output value is HIGH (5V) at that particular pin. The circuit diagram for the recipe is shown in *Figure 4.4*:

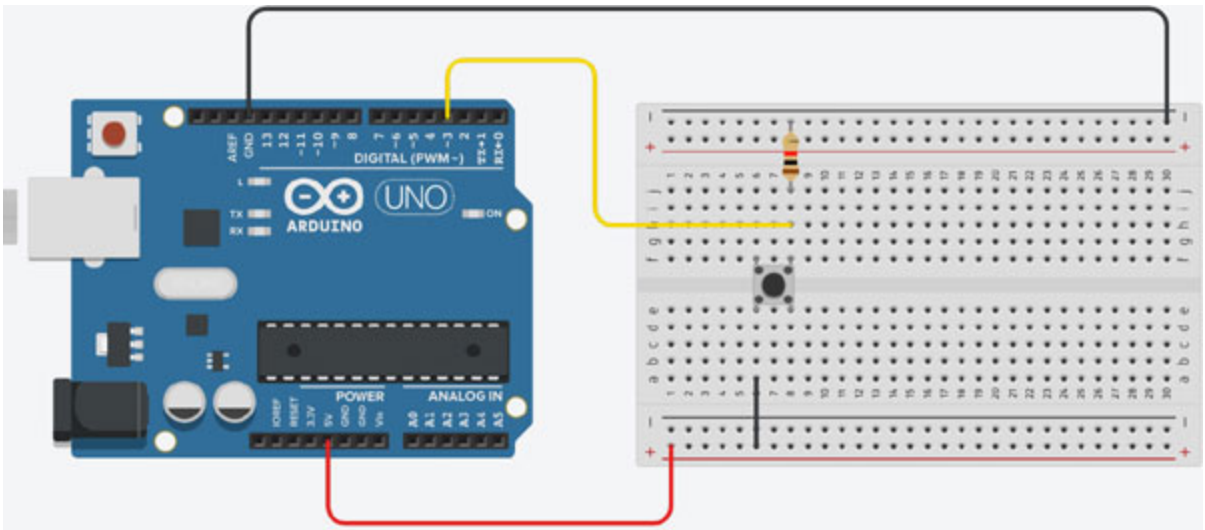


Figure 4.4: *Connections of a push button with an Arduino board*

The sketch for the recipe is given as follows.

```
int time1;
int time2;
int duration;
int Run_timer;
void setup(){
pinMode (3, INPUT);
Serial.begin(9600);
}

void loop(){
 if (Run_timer == 0 && digitalRead(3) == LOW)
```

```
  {
    time1 = millis();
    Run_timer = 1;
    }
 if (Run_timer == 1 && digitalRead(3) == HIGH)
  {
    time2 = millis();
    Run_timer = 0;
    duration = time2 - time1;
    Serial.print (“The button was pressed for (milliseconds):
    “);
    Serial.println (duration);
 }
}
```

In this sketch, we have defined **`time1`** and **`time2`** to store the starting and ending time values to measure the **`duration`** for which the button was pressed. Duration is the difference between these two-time values in milliseconds. We have also declared a flag variable **`Run_timer`** to detect the running status of the timer.

Recipe 3: Generating red, green, and blue colors using an RGB LED

A **Light Emitting Diode** (**LED**) emits one color if an input voltage greater than the threshold voltage of its PN junction is provided in a forward biased configuration. Every LED emits a single unique color when it is forward-biased. So, we must use different LEDs to display different colored lights. A **Red, Green, Blue** (**RGB**) LED is a solution to provide multiple colors with a single LED. In this LED, we have 4 pins (*Figure 4.5*):

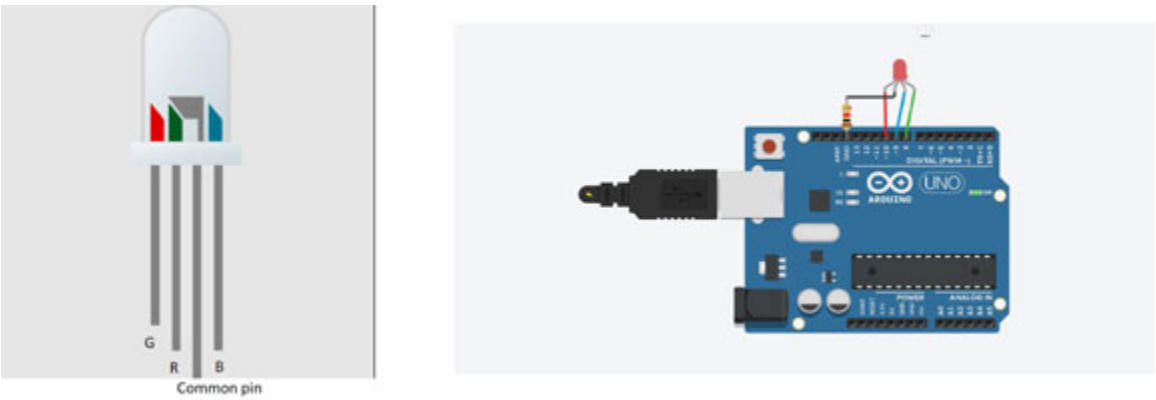


Figure 4.5: *Pins of an RGB LED and its connection with an Arduino board*

One pin is common and connected to the ground. There are three other pins for the three primary colors, that is, Red, Green and Blue. If we supply a HIGH signal to any of these colors, we can generate the particular color on this LED.

In this recipe, we have connected the RGB LED's common pin to the Arduino UNO's ground pin. Three color pins are connected to digital pins 8, 9, and 10. We will control the output of each of these pins with the help of three different keys on our computer's keyboard. We will assign numbers 1 for **`Red`**, 2 for **`Green`**, and 3 for **`Blue`**, and we will use the concept of serial communication between the computer and Arduino to provide input commands from the keyboard to the Arduino board.

```
int in1 = 0;
int Red =10;
int Green = 8;
int Blue = 9;
void setup(){
  pinMode(Red, OUTPUT);
  pinMode(Green, OUTPUT);
  pinMode(Blue, OUTPUT);
  Serial.begin(9600);
}

void loop(){
  if (Serial.available()>0) {
    in1 = Serial.read();
```

```
    if (in1 == 49){
    digitalWrite(Red, HIGH);
    delay(500);
    }
    else
    {
    digitalWrite(Red, LOW);
    }
    if (in1 == 50){
    digitalWrite(Green, HIGH);
    delay(500);
    }
    else
    {
    digitalWrite(Green, LOW);
    }
    if (in1 == 51){
    digitalWrite(Blue, HIGH);
    delay(500);
    }
    else
    {
    digitalWrite(Blue, LOW);
    }
  }
}
```

Recipe 4: Generating millions of random colors using RGB LED

An RGB LED is quite a unique display device. In the previous recipe, we observed that we could generate three primary colors by pressing three different buttons on our keyboard. In that recipe, we did not exploit the full potential of the RGB LED. An RGB LED can generate not 3 but 2^24 = 16,777,216 (more than 16 million) colors. Now, you must be wondering

how we can generate such a massive number of colors using a single device that has such simple construction and operation.

We know that red, green and blue are the three primary colors, and every other color can be made by using an appropriate mixture of these colors.

Figure 4.6 shows the RGB spectrum. Every color value is stored in digital representation as an 8-bit binary code. That means every color value can take a value between 0 to 255. These values define the intensity of that particular primary color. 0 represents absence, and 255 represents maximum presence. Now, for a colored image, we choose the combination of appropriate intensities of three primary colors. Therefore, to make any color, we have three 8-bit values, which implies that any color information requires 24 bits. This is why we can have 2^24 = 16,777,216 different colors in digital representation. Refer to the following figure:

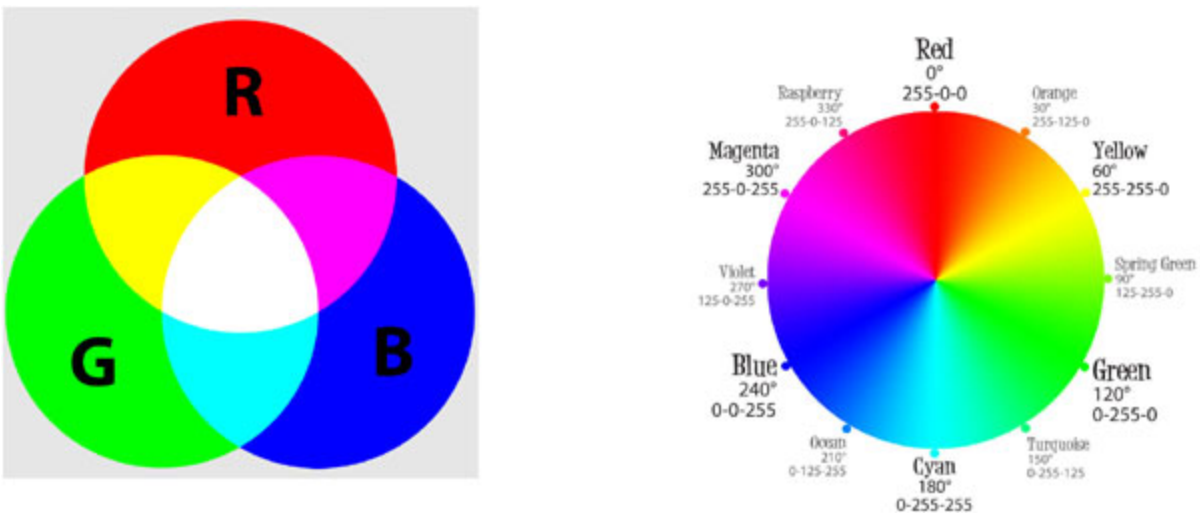


Figure 4.6: *The RGB spectrum and formation of new colors using the spectrum*

Representing a color on a digital display (Computer or mobile screen) is done with millions of RGB LEDs. If we provide all the pins with less or more intensity at the input pins of three different color pins of the RGB LEDs, we can generate these new colors. Therefore, we should provide its corresponding combination as the input to generate any new color on an RGB LED. For example, refer to Figure 4.6, the code for yellow is 255-255-0.

In the previous chapter, we learned the concept of PWM outputs. We will use the same concept to generate these color codes for three different pins.

We must ensure that all three pins are connected to the PWM pins of the Arduino board. The circuit diagram for the recipe is shown in *Figure 4.7*:

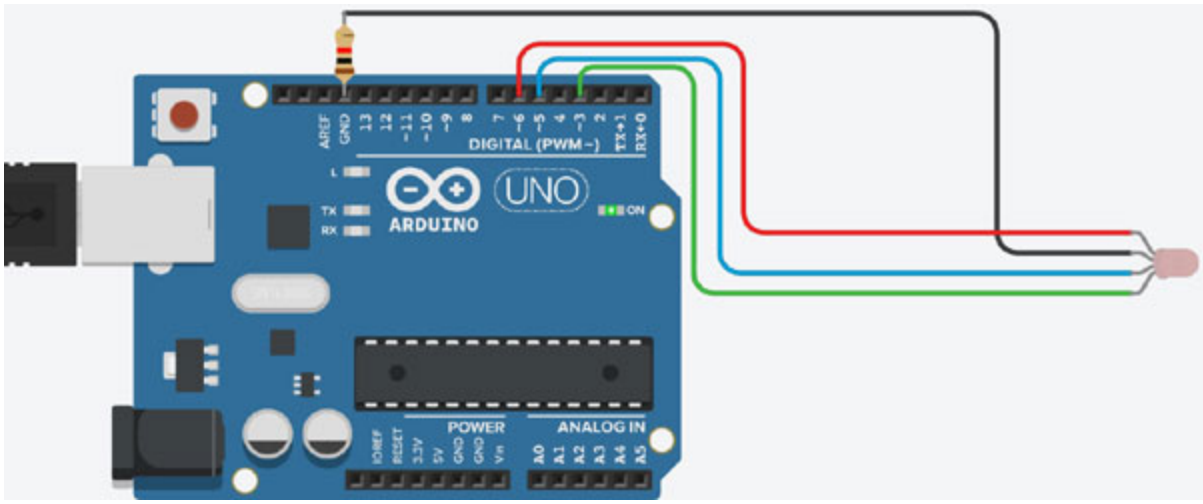


Figure 4.7: *Circuit diagram to generate millions of colors using the RGB LED*

The sketch for the recipe is as follows:

```
int Red_pin =6;
int Green_pin = 3;
int Blue_pin = 5;
int Red, Green, Blue = 0;
void setup(){
 pinMode(Red_pin, OUTPUT);
 pinMode(Green_pin, OUTPUT);
 pinMode(Blue_pin, OUTPUT);
}

void loop(){
 Red = random(0,256);
 Green = random(0,256);
 Blue = random(0,256);
 analogWrite(Red_pin,Red);
 analogWrite(Green_pin,Green);
 analogWrite(Blue_pin,Blue);
 delay(500);
}
```

Recipe 5: Driving a 7-segment LED display

In this recipe, we will discuss a new display device. When we display multiple outputs, we can use multiple LEDs. For example, if we have to display 5 different numbers, we can use 5 different LEDs. Suppose we have to display 15 different numbers, then we will fall short of the digital pins on our Arduino board. In such cases, we have to use some special display devices. One of such display devices is a 7-segment display, as shown in *Figure 4.8*:

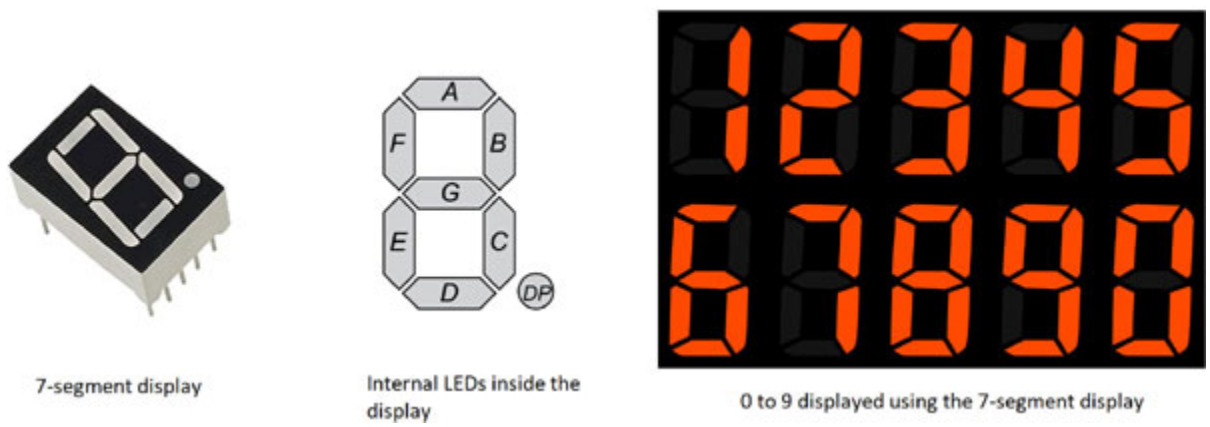


Figure 4.8: *A 7-segment display displaying numbers using the same*

A 7-segment display is designed by using 7 different LEDs in a specific manner. *Figure 4.8* shows the physical structure of the 7-segment display. This display device has 10 pins 5 on each side of the device. The display is designed by using 7 leads by making a shape of 8. In the same figure, we can observe how can the different numbers be generated using a single display. For example, if you want to display 0, then we'll switch off the name **G** and switch on all other LEDs.

Similarly, we can observe that LEDs **B** and **C** are turned-on to display 1, and all leads are on to display 8. Using the same display, we can even display some other symbols that may not be recognized in our daily routine. The total number of symbols that can be generated using this display is 2^7, which is equal to 128.

To display any number using a 7-segment display from Arduino, we need only 7 output pins. *Figure 4.9* shows the circuit diagram connecting a 7-segment display with the Arduino board. We must ensure that we provide

correct connections for every pin and do not forget to include the resistors. Refer to the following figure:

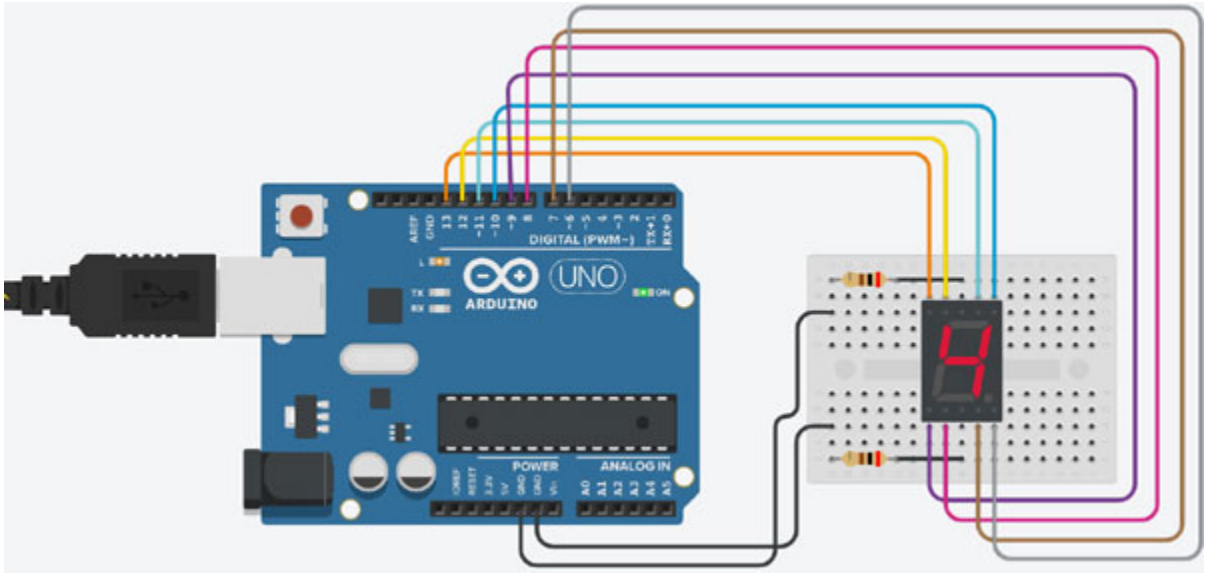


Figure 4.9: *Circuit connections of a 7-segment display with an Arduino board*

For programming, we will define 10 different functions to display 10 different digits. For example, if we have to display 0, then we will store it as 1111110. This implies that LEDs A, B, C, D, E and F are at logic 1 (HIGH) and LED G is at logic 0 (LOW). Similarly, the code for displaying 1 will be 0110000, and so on. Therefore, in every respective function, we will define the LED outputs. In the **`loop()`** function, we call these functions one after another to display all the digits in a sequence. The sketch of this recipe is a little longer as it contains multiple functions. Let us understand the sketch in shorter parts now. Initially, we define the setup function, where we declare digital pins 6 to 13 as output pins:

```
void setup(void)
{
  pinMode(13, OUTPUT);
  pinMode(12, OUTPUT);
  pinMode(11, OUTPUT);
  pinMode(10, OUTPUT);
  pinMode(9, OUTPUT);
  pinMode(8, OUTPUT);
  pinMode(7, OUTPUT);
```

```
  pinMode(6, OUTPUT);
}
```

Now, we will define the glowing patterns for 7 LEDs corresponding to the different digits. For example, to display 0, we need to keep LED G off and all other ON. Therefore, PIN 13 is set to LOW, and all others are set to HIGH. The same is coded in the **`zero()`** function as follows:

```
void zero(void) {
  digitalWrite(13, LOW);
  digitalWrite(12, HIGH);
  digitalWrite(11, HIGH);
  digitalWrite(10, HIGH);
  digitalWrite(9, HIGH);
  digitalWrite(8, HIGH);
  digitalWrite(7, HIGH);
  digitalWrite(6, LOW);
}
```

The next function shows the glowing pattern for displaying 1:

```
void one(void) {
  digitalWrite(13, LOW);
  digitalWrite(12, LOW);
  digitalWrite(11, LOW);
  digitalWrite(10, HIGH);
  digitalWrite(9, LOW);
  digitalWrite(8, LOW);
  digitalWrite(7, HIGH);
  digitalWrite(6, LOW);
}
```

The **`two()`** function describes the pin values for displaying number 2:

```
void two(void) {
  digitalWrite(13, HIGH);
  digitalWrite(12, LOW);
  digitalWrite(11, HIGH);
  digitalWrite(10, HIGH);
```

```
  digitalWrite(9, HIGH);
  digitalWrite(8, HIGH);
  digitalWrite(7, LOW);
  digitalWrite(6, LOW);
 }
```

The **`three()`** function is written for display pattern for displaying 3:

```
 void three(void) {
  digitalWrite(13, HIGH);
  digitalWrite(12, LOW);
  digitalWrite(11, HIGH);
  digitalWrite(10, HIGH);
  digitalWrite(9, LOW);
  digitalWrite(8, HIGH);
  digitalWrite(7, HIGH);
  digitalWrite(6, LOW);
 }
```

The **`four()`** function is written for display pattern for displaying 4:

```
 void four(void) {
  digitalWrite(13, HIGH);
  digitalWrite(12, HIGH);
  digitalWrite(11, LOW);
  digitalWrite(10, HIGH);
  digitalWrite(9, LOW);
  digitalWrite(8, LOW);
  digitalWrite(7, HIGH);
  digitalWrite(6, LOW);
 }
```

The **`five()`** function is written for display pattern for displaying 5:

```
 void five(void) {
  digitalWrite(13, HIGH);
  digitalWrite(12, HIGH);
  digitalWrite(11, HIGH);
  digitalWrite(10, LOW);
```

```
  digitalWrite(9, LOW);
  digitalWrite(8, HIGH);
  digitalWrite(7, HIGH);
  digitalWrite(6, LOW);
 }
```

The **`six()`** function is written for display pattern for displaying 6:

```
 void six(void) {
  digitalWrite(13, HIGH);
  digitalWrite(12, HIGH);
  digitalWrite(11, HIGH);
  digitalWrite(10, LOW);
  digitalWrite(9, HIGH);
  digitalWrite(8, HIGH);
  digitalWrite(7, HIGH);
  digitalWrite(6, LOW);
 }
```

The **`seven()`** function is written for display pattern for displaying 7.

```
 void seven(void) {
  digitalWrite(13, LOW);
  digitalWrite(12, LOW);
  digitalWrite(11, HIGH);
  digitalWrite(10, HIGH);
  digitalWrite(9, LOW);
  digitalWrite(8, LOW);
  digitalWrite(7, HIGH);
  digitalWrite(6, LOW);
 }
```

The **`eight()`** function is written for display pattern for displaying 8. We can observe that all the pins are set to HIGH in this function:

```
 void eight(void) {
  digitalWrite(13, HIGH);
  digitalWrite(12, HIGH);
  digitalWrite(11, HIGH);
```

```
  digitalWrite(10, HIGH);
  digitalWrite(9, HIGH);
  digitalWrite(8, HIGH);
  digitalWrite(7, HIGH);
  digitalWrite(6, LOW);
}
```

The **`nine()`** function is written for display pattern for displaying 9:

```
void nine(void) {
  digitalWrite(13, HIGH);
  digitalWrite(12, HIGH);
  digitalWrite(11, HIGH);
  digitalWrite(10, HIGH);
  digitalWrite(9, LOW);
  digitalWrite(8, HIGH);
  digitalWrite(7, HIGH);
  digitalWrite(6, LOW);
}
```

Now, the **`loop()`** function is written. In this, we call all the preceding 10 functions one after another to display all numbers in a sequence:

```
void loop(void)
{
  zero();
  delay(1000);
  one();
  delay(1000);
  two();
  delay(1000);
  three();
  delay(1000);
  four();
  delay(1000);
  five();
  delay(1000);
  six();
```

```
  delay(1000);
  seven();
  delay(1000);
  eight();
  delay(1000);
  nine();
  delay(1000);
}
```

Recipe 6: Displaying a blinking message on a 16x2 LCD

A 7-segment display can display only one character at a time. In real-world applications, we need to display words with multiple characters simultaneously. Therefore, a 7-segment display might not be sufficient to display such outputs. For this purpose, we use a LCD unit. Using an LCD, we can display multiple characters at any time. Refer to the following table:

Pin name	Purpose	Connected to
VSS	VSS stands for the ground pin	The ground terminal of the Arduino
VDD	VDD stands for the power supply	Connected to 5V terminal
VEE	This is the control pin to control the brightness of the LCD	Connected to the central pin of a potentiometer
RS	Register Select pin let us decide between Command/Data Register	Set the value to 0 for command mode and 1 for the data register
RW	Read/Write pin helps in deciding whether to read or write on the LCD	Set 0 to write and 1 to read; mostly we keep it 0
E	Enable pin to enable the whole operation	Mostly kept as 1
D0 to D7	8 data pins to transmit data between Arduino and LCD	Only 4 pins (D4 to D7) are used for writing purposes

Table 4.1: *PIN names and their purposes for a 16x2 LCD*

Figure 4.10 shows a 16x2 LCD. We can observe that there is a blank panel in the center and various pins for connections. The names and their respective connections are also shown in the same figure.

The pins and their purpose are explained in *Table 4.1*. Let us understand the purposes of all these pins. These pins' names are mentioned per the names mentioned in *Figure 4.10*:

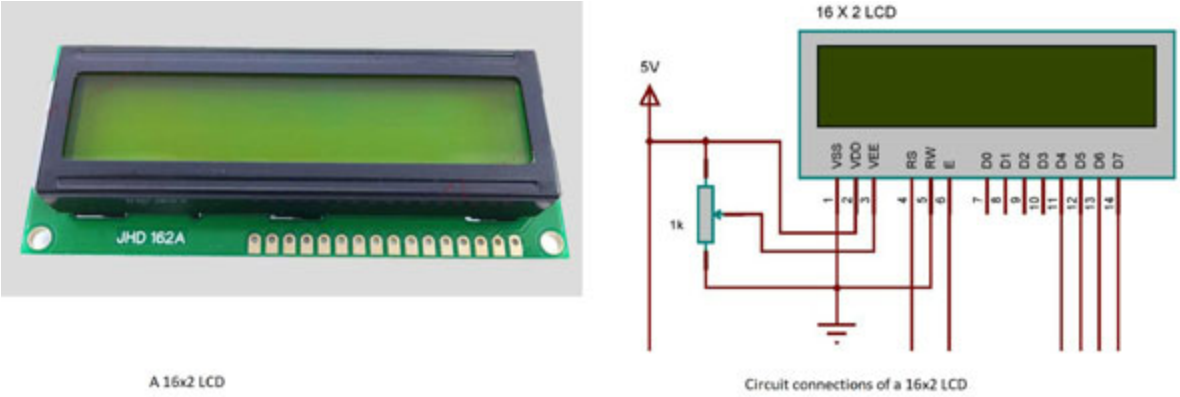


Figure 4.10: *A 16x2 LCD and its pin connections*

To display anything using the 16x2 LCD, we need sophisticated symbol declarations and the right combination of control pins. To help us with all such complex coding, a library is already defined. We must include the **`LiquidCrystal.h`** library to access all such predefined functions and patterns.

The most important part of the sketch is to define the pin connections of the LCD with the Arduino board. The connection of a 16x2 LCD is shown in *Figure 4.11*:

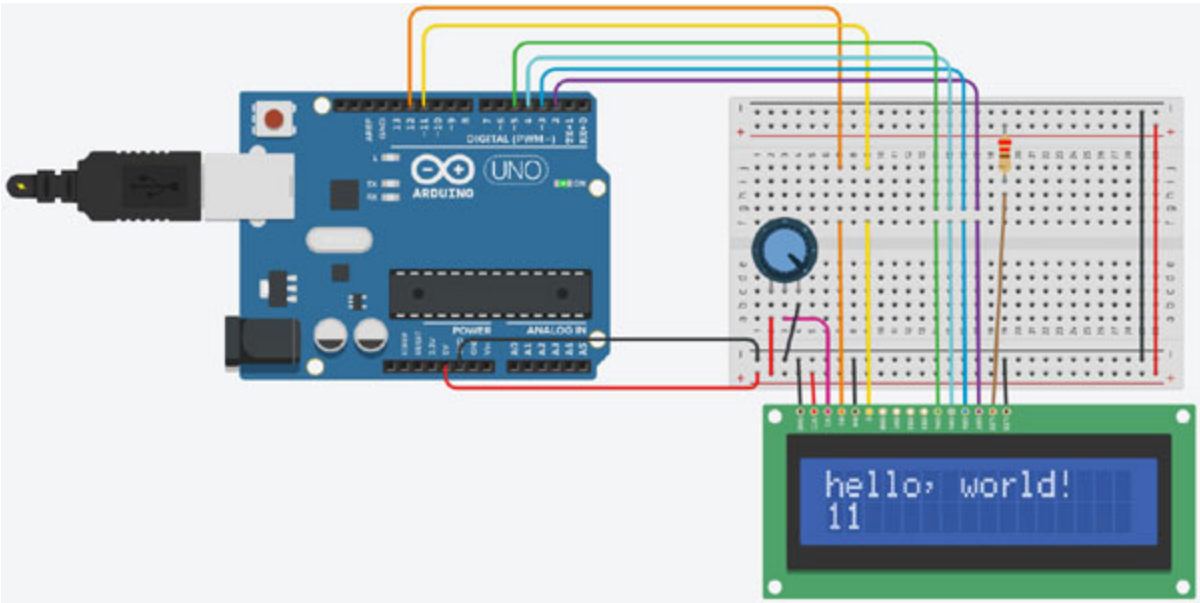


Figure 4.11: *Connecting a 16x2 LCD to Arduino and displaying "Hello world!" along with the time*

We must declare the connection of 6 pins in the same order. The order of the pin is (RS, E, D4, D5, D6 D7). In this figure, we have connected RS to PIN 12, E to 11, D4 to 5, D5 to 4, D6 to 3, and D7 to 2. So, the command to establish this connection is as follows:

```
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
```

The rest of the coding is quite simple and can be done using the inbuilt functions of the LCD library.

The sketch for the recipe is as follows:

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
void setup() {
  lcd.begin(16, 2);
  lcd.print("hello, world!");
}
void loop() {
  lcd.setCursor(0, 1);
  lcd.print(millis() / 1000);
}
```

The **`setup()`** function is simple. We defined the LCD size (16x2) and then used the **`lcd.print`** function of the lcd library to print "**`hello world!`**". We can replace it with any other string (length less than 16) and display it easily.

In the loop function, we must understand the **`lcd.setCursor`** function. This LCD has 16 columns and 2 rows. So, the index positions of rows are 0 and 1 (0 for the first row and 1 for the second). Similarly, the index positions for the columns are from 0 to 15. We have set the cursor position to (0,1), which means the first column of the second row. We displayed the time passed in seconds (milliseconds divided by 1000) at this location. You must try changing the column value and observe the change. While connecting the LCD, we must ensure that the potentiometer provides 5V to VEE.

Recipe 7: Displaying a left scrolling message on a 16x2 LCD

In the previous recipe, we learned the process to display a message on a 16x2 LCD, but we faced a hurdle while doing so. If you go back to the same recipe and try to display any message with a length of more than 16 characters, you will observe that the message gets truncated after 16 characters. So, is it impossible to display any message longer than 16 characters using a 16x2 LCD? Well, the answer is a big no. You must have seen large LED-based billboards around you. All these boards have scrolling messages. The same can be implemented on the 16x2 LCD as well.

We can use the scroll function and scroll the message towards the left or right on a 16x2 LCD. **`scrollDisplayLeft()`** and **`scrollDisplayRight()`** are used to scroll the message left or right. The connections remain the same for displaying a scrolling message. The connections are shown in *Figure 4.12*:

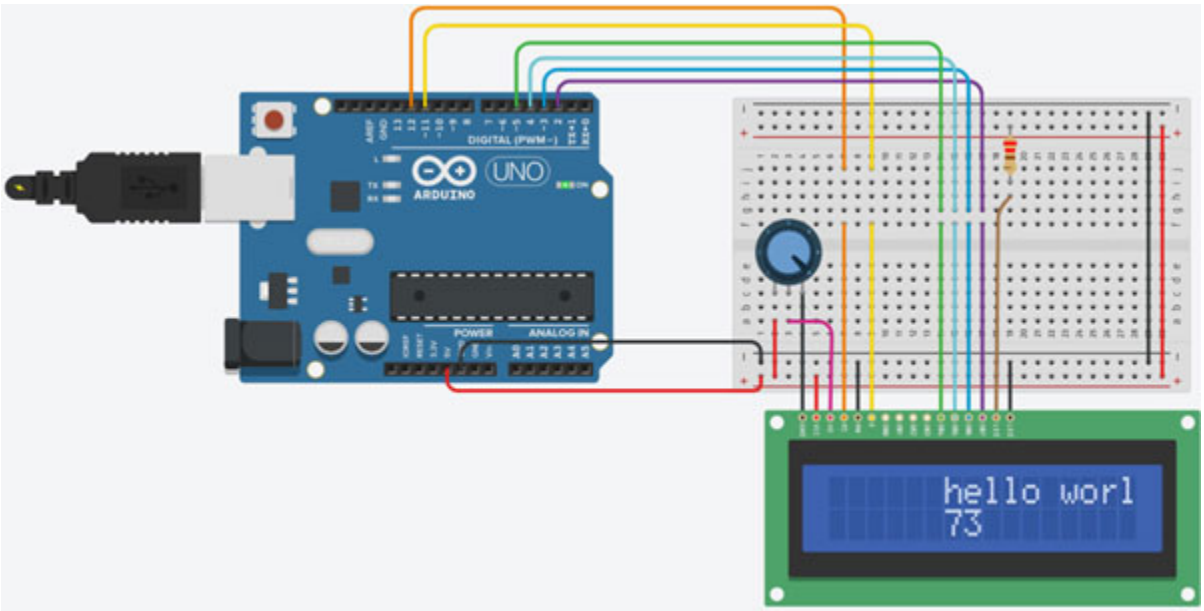


Figure 4.12: *Scrolling the text towards the left on a 16x2 LCD*

There is a minor change in the program. The sketch for the same is given as follows:

```
#include <LiquidCrystal.h>
```

```
int seconds = 0;
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
void setup()
{
 lcd.begin(16, 2);
 lcd.print(“hello world!”);
}

void loop()
{
 lcd.setCursor(0, 1);
 lcd.scrollDisplayLeft();
 lcd.print(seconds);
 delay(1000);
 seconds += 1;
}
```

Recipe 8: Displaying a rotating sequence using a NeoPixel LED

NeoPixels are a special type of RGB LEDs that are arranged on a strip. Multiple LEDs are connected in a straight line or circular manner. All these can be separately programmed. The biggest advantage of using NeoPixels is that all these LEDs can be programmed using a single pin of Arduino or any other microcontroller.

Figure 4.13 shows some of the commonly available NeoPixels in the market:

A circular 8-bit neopixel A strip of 8 neopixels A 16 bit circular neopixel

Figure 4.13: Different types of NeoPixels available for displaying patterns

NeoPixel library is developed by AdaFruit (a renowned company in hardware development). We can include the functions of the library by calling the following:

```
#include <Adafruit_NeoPixel.h>.
```

If the library is not available in IDE, then install using the library manager.

We will use the functions related to the library to generate different patterns. In this recipe, we connect the input pin of NeoPixel to pin 6 of Arduino. If we want to attach more NeoPixels, we can connect the output pin of the first NeoPixel to the input of the next. Otherwise, the output pin is left blank. The ground and power pins of NeoPixels are connected to the ground and 5V of the Arduino board.

The circuit for the recipe is shown in *Figure 4.14*:

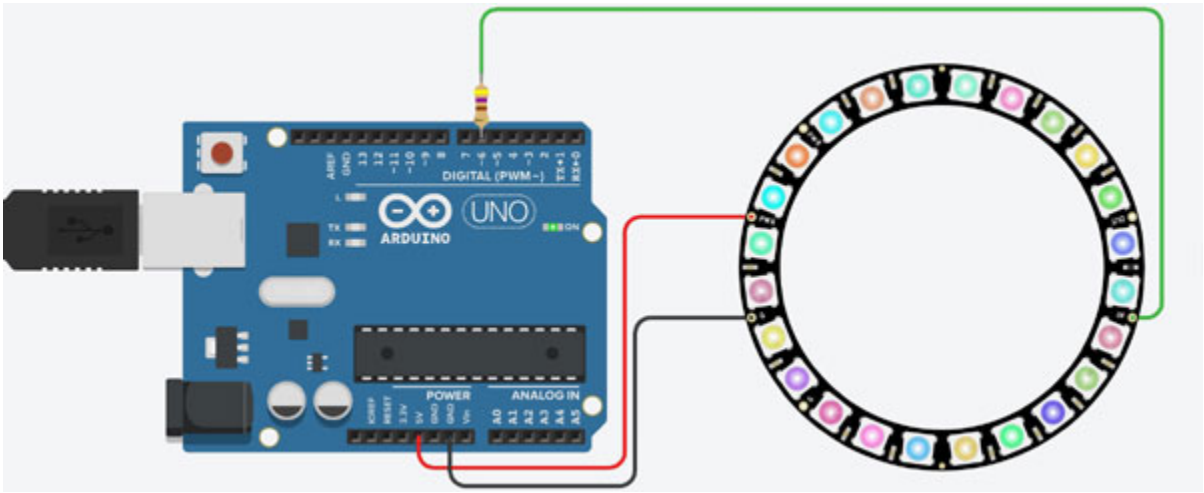


Figure 4.14: Displaying a circular pattern on 24-bit NeoPixel with Arduino

The sketch for the recipe is as follows:

```
#include <Adafruit_NeoPixel.h>
#define PIN 6
byte R, G, B;
Adafruit_NeoPixel strip = Adafruit_NeoPixel(24, PIN, NEO_GRB +
NEO_KHZ800);
void setup()
```

```
{
 strip.begin();
 for(byte i=0; i<24; i++)
  strip.setPixelColor(i, 255, 0, 255);
 strip.show();
 randomSeed(analogRead(0));
}
void loop()
{
 for(byte i=0; i<24; i++)
 {
  R=random(256);
  G=random(256);
  B=random(256);
  strip.setPixelColor(i, R, G, B);
 }
 strip.show();
 delay(1000);
}
```

In this sketch, we have used the NeoPixel library. From this library, we have used the declaration function and defined that at PIN 6, we have connected 24 RGB NeoPixel LEDs at 800kHz. In the **`setup()`** function, we set the color of all the 24 LEDs to (255, 0, 255). Therefore, all 24 LEDs will display a common color. In the **`loop()`** function, we have defined random values for red, green, and blue (R, G and B) and then allocated these random values to all 24 LEDs. The process is repeated every second, and we can observe random new colors on every lead.

Recipe 9: Generating different voltage values using a potentiometer and display them on an LCD

We have discussed switches and display devices in the earlier recipes of this chapter. Now, we will integrate and design some useful applications. In this recipe, we will take input from a potentiometer and display the voltage generated by the potentiometer on a 16x2 LCD.

A potentiometer is connected between 0 to 5V signals. Therefore, it can generate any voltage signal between 0 and 5V. We read this analog voltage at one of Arduino's analogue pins (A0 is this recipe). As discussed in the programming chapters, the integer values read at analog pins are in the range of 0 to 1023. Therefore, the values of 0V maps to 0 and the value of 5V maps to 1023. Now, we have to convert this integer value into the respective voltage value and store it in a variable (**`temp`** in this recipe). We will display the variable value on the LCD.

The circuit diagram for the recipe is shown in *Figure 4.15*. We can observe that we have used the potentiometer and a multimeter to display the voltage generated by the potentiometer. The displayed voltage on the multimeter is independent of Arduino. So, we can say that this is the voltage generated by the potentiometer. The input of the potentiometer is taken to analog pin A0 and converted to the voltage through the programming and displayed on the LCD. We can observe in the same figure that the voltage value displayed on the LCD matches the value on the multimeter. Refer to the following figure:

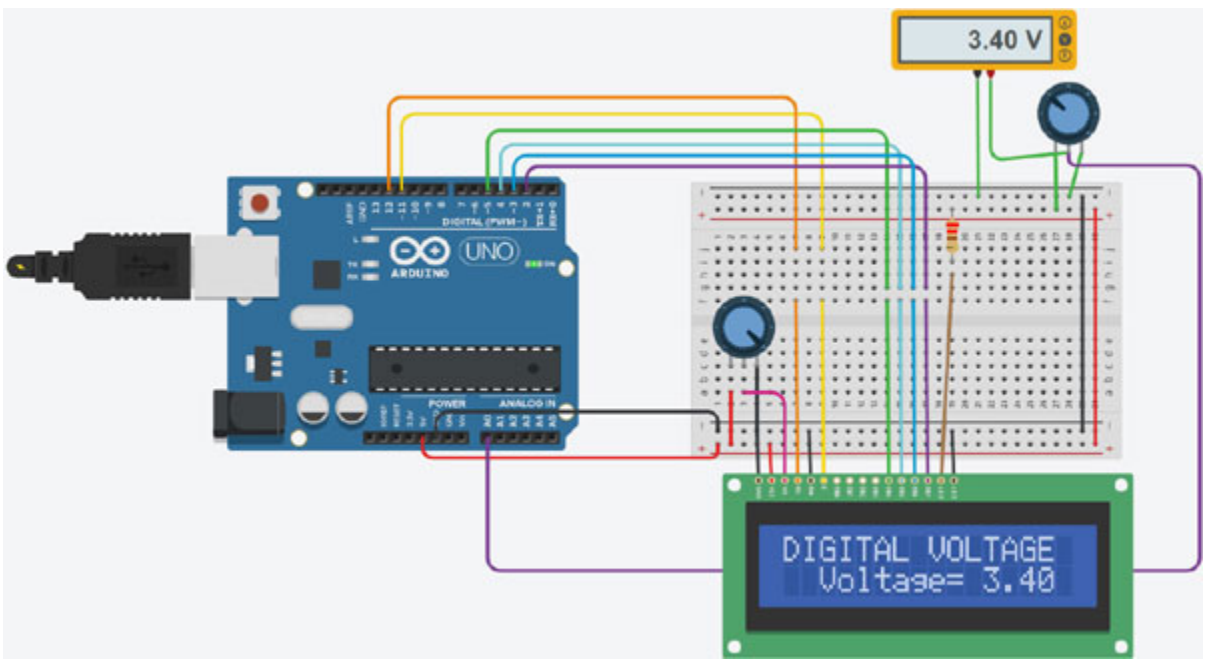


Figure 4.15: *A digital voltmeter designed using an Arduino and LCD*

The sketch for the recipe is as follows:

```
#include <LiquidCrystal.h>
float temp=0.0;
```

```
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
void setup()
{
 lcd.begin(16, 2); // set up the LCD's number of columns and
 rows:
 lcd.print("DIGITAL VOLTAGE");
}
void loop()
{
   int analog_value = analogRead(A0);
   temp = (analog_value * 5.0) / 1023.0;
   lcd.setCursor(2, 1);
   lcd.print("Voltage= ");
   lcd.print(temp);
   delay(300);
}
```

Recipe 10: Design a 6-level water level indicator using an LED array

We have a water tank, and it is situated on the top of our home/building. If we need to check the water level in the tank at any given time, we must go to the top, open the tank and then check the level. This can be quite risky and inconvenient since the tank is situated quite high. In this recipe, lets us design a simple circuit that can perform this task for us.

We will dip a plastic pipe inside the tank with 6 metal connections at regular intervals. Therefore, the whole water depth is divided into these 6 levels. As soon as the water touches any of these metal contacts, the circuit will be completed (because water is a conductor), and we will get a signal from the metal contact. Now, we connect 6 LEDs inside the room in a row. Each of these LEDs depends on the signal value from the metal contacts. Therefore, if only one metal contact is active, the first LED will glow. If the next metal contact is active, the second LED will also glow, and so on. On a full tank, we will be having all the LEDs glowing. This is quite a simple

system, and we will be able to estimate the water level while sitting inside our room.

The circuit for the recipe is shown in *Figure 4.16*. This figure shows that PINs 2, 3, 4, 5, 6, and 7 are the input pins. The input is taken from slider switches. Since the metal contacts could not be shown in the simulation environment, these switches are used. In a real implementation, we will use metal contacts instead of switches. PINs 8, 9, 10, 11, 12 and 13 are the output pins that control 6 different LEDs. The input of pin 2 will control the LED at PIN 13; PIN 3 will control PIN 12, and so on. Refer to the following figure:

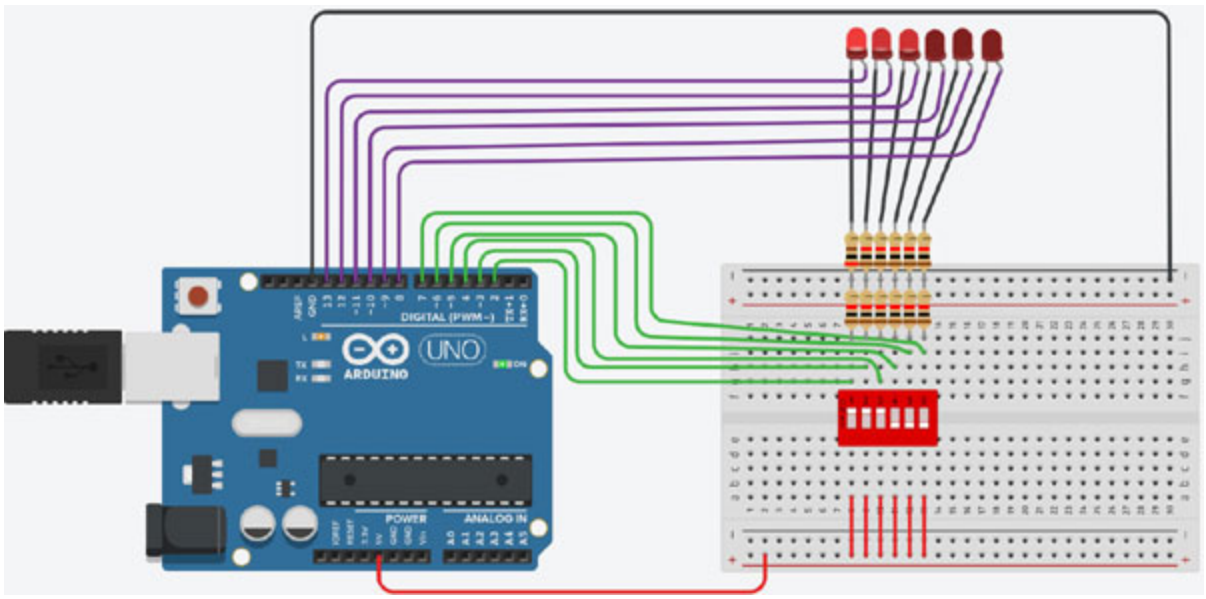


Figure 4.16: *The circuit diagram for a water level indicator system*

The sketch for the recipe is as follows:

```
void setup()
{
  pinMode(2,INPUT);
  pinMode(3,INPUT);
  pinMode(4,INPUT);
  pinMode(5,INPUT);
  pinMode(6,INPUT);
  pinMode(7,INPUT);
  pinMode(8,OUTPUT);
```

```
  pinMode(9,OUTPUT);
  pinMode(10,OUTPUT);
  pinMode(11,OUTPUT);
  pinMode(12,OUTPUT);
  pinMode(13,OUTPUT);
}
void loop()
{
  digitalWrite(13, digitalRead(2));
  digitalWrite(12, digitalRead(3));
  digitalWrite(11, digitalRead(4));
  digitalWrite(10, digitalRead(5));
  digitalWrite(9, digitalRead(6));
  digitalWrite(8, digitalRead(7));
  delay(100); // Wait for 1000 millisecond(s)
}
```

Conclusion

In this chapter, we explored two types of devices that can be connected to an Arduino board. We learnt how to connect a switch or a button to the Arduino board and provide different digital inputs at the digital pins. These switches can control different operations through the Arduino board. Then, we discussed the display devices, which are categorised as output devices. We can display a digital value using an LED or 7-segment display. We understood their connections and the mistakes to be avoided while connecting them. Then, we looked at some advanced display devices like LCD and OLED. We were able to display strings using the same. In this exploration of display devices, we also got to know about some fancy display devices like NeoPixels. These devices can be used to generate some interesting patterns with programming.

In the next chapter, we will learn about the motor. We will discuss the different types of motors and their connections with the Arduino board. Some of the motors can be directly controlled by the Arduino, while a few

require additional drivers. We'll also develop some simple robots using these motors.

Points to remember

- While connecting a switch, make sure that a resistor is always connected at one end. It will avoid the direct connection between ground and supply voltage and save the circuit.
- An LED must have a series resistor to control the flow of current and save it from burning.
- The LCD screen's brightness should be adjusted to the maximum in the beginning.
- The LCD's connections must be checked properly (specially VCC and ground) before switching it on.
- Soldering of NeoPixels should be done carefully.
- The potentiometer must be properly tuned in LCD.

Multiple choice questions

1. **A common Cathode 7-segment display is different from a common anode LED in terms of which of the following?**
    a. The value of the resistor should be different
    b. All LED pins are connected to the ground in both displays
    c. All LED pins are connected to the ground in the common cathode and VCC in the common anode.
    d. All LED pins are connected to VCC in the common cathode and ground in the common anode.
2. **What is the resistance of an open switch?**
    a. Very low
    b. Very high
    c. Exactly 100kOhms
    d. Keep on changing

3. **How many terminals does a potentiometer have?**
    a. 1
    b. 2
    c. 3
    d. 4
4. **The message on a 16x2 LCD can be scrolled in which direction?**
    a. None
    b. Left
    c. Right
    d. Both left and right
5. **A NeoPixel requires ___ pin(s) for programming apart from ground and power pins.**
    a. 1
    b. 2
    c. 3
    d. 4

Answers

1. **c**
2. **b**
3. **c**
4. **d**
5. **a**

CHAPTER 5

Sensor Integration With Arduino

Introduction

In this section, we will focus on hardware integration to an Arduino board. In the previous chapter, we discussed the integration of various switches and display devices with an Arduino board. These two are essential input and output devices. The next category of input devices is sensors. There are various sensors to sense natural and artificial properties in our surroundings. Sensors are used to detect the presence/absence or the quantity of different physical inputs. The specific input could be light, pressure, concentration, heat, motion, or any other physical property.

In this chapter, we will learn the process of integrating different sensors with an Arduino board and read different kinds of input values from them. We'll start with detecting light intensity with a **Light Dependent Resistor** (**LDR**). Then, we will discuss an Infra-Red (IR) proximity sensor and ultrasonic sensor for measuring distances. Next, we'll learn how to connect a microphone and take sound as an input signal to an Arduino board. We'll learn about the working and applications of gas sensors. We will then discuss how we can detect any motion using a **Passive Infra-Red** (**PIR**) sensor and make a robust burglar alarm. The last input device in this chapter will be a 4x4 keypad to provide numerical input to the Arduino board. We will develop a few interesting recipes using all these sensors.

Structure

This chapter will have the following recipes related to different sensors with an Arduino board:

- Detecting the intensity of light
- Detecting the object with an IR proximity sensor
- Measuring the distance of a wall from the sensor

- Clap-controlled home appliance
- Designing a water level indicator by measuring the depth of the water level
- Detecting the LPG leakage in a home
- Detecting the motion of a person around your home
- Designing a robust burglar alarm system
- Securing access to a door using a 4-digit password
- Designing a multi-password door lock
- Designing a reverse car parking assistant system
- Measuring the temperature of the surroundings
- Designing a digital weather station to show real-time weather conditions

Objectives

By the end of this chapter, the learner will be able to do the following:

- Differentiate between analog and digital sensors
- Connect a variety of sensors properly to an Arduino board
- Tune different sensors to obtain correct input values
- Troubleshoot a circuit having single or multiple sensors
- Design own solutions for real-world problems using sensors

Recipe 1: Detecting the intensity of light

We are exposed to natural or artificial light whenever we are awake. Sometimes we are in well-lit areas, and sometimes in low-lit areas. Our eyes are natural sensors to detect light intensity in our surroundings. We perform many activities due to the presence of light around us. It becomes very difficult to work in the absence of proper lighting.

In this recipe, we will learn about a sensor that can measure the intensity of light in the surroundings and design an automatic streetlight. A **Light-Dependent Resistor** (**LDR**) is a special resistor designed to measure the intensity of light falling on it. It is also known as a photoresistor, a

Photocell, or photo-conductive cell. The surface of an LDR is made of cadmium sulfide (CdS). The resistance of this compound decreases with the increase in light intensity.

In *Figure 5.1*, we can see different sizes of LDR on a millimeter scale. The next two sections in the figure show the observed resistance of LDR under minimum and maximum light intensity. In the last section of the figure, we have attached a resistor in series with the LDR and made a voltage divider circuit with the help of a 9V battery. The output at the central pin can be a voltage between 0 to 9V, depending on the light intensity falling on the LDR.

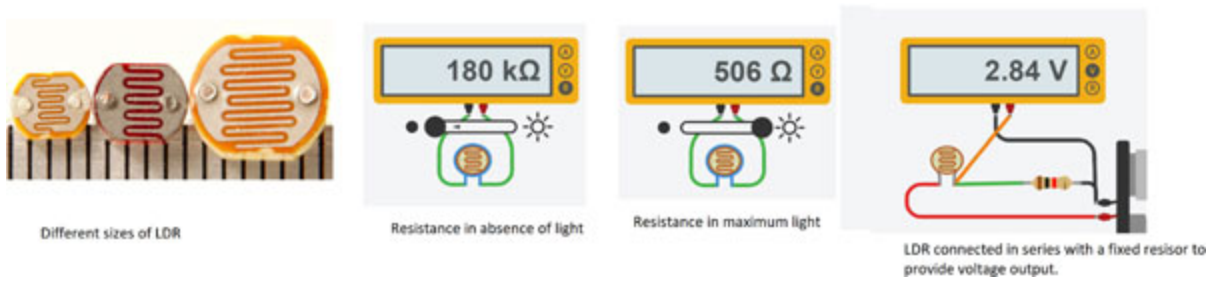


Figure 5.1: *Different types of LDR and the process of obtaining voltage output from LDR*

We have learnt that we can generate different voltage values by changing the light's intensity on an LDR's surface. Let us develop the final part of this recipe now. In this section, we will design an automatic streetlight. An Arduino board will control this streetlight. The board will take the input from an LDR, and the lights (LED) will be turned on whenever darkness is detected. In the morning, whenever the intensity of light increases, the light will be turned off. We will use **`if-else`** statements in the coding. The circuit for the automatic streetlight system is shown in *Figure 5.2*:

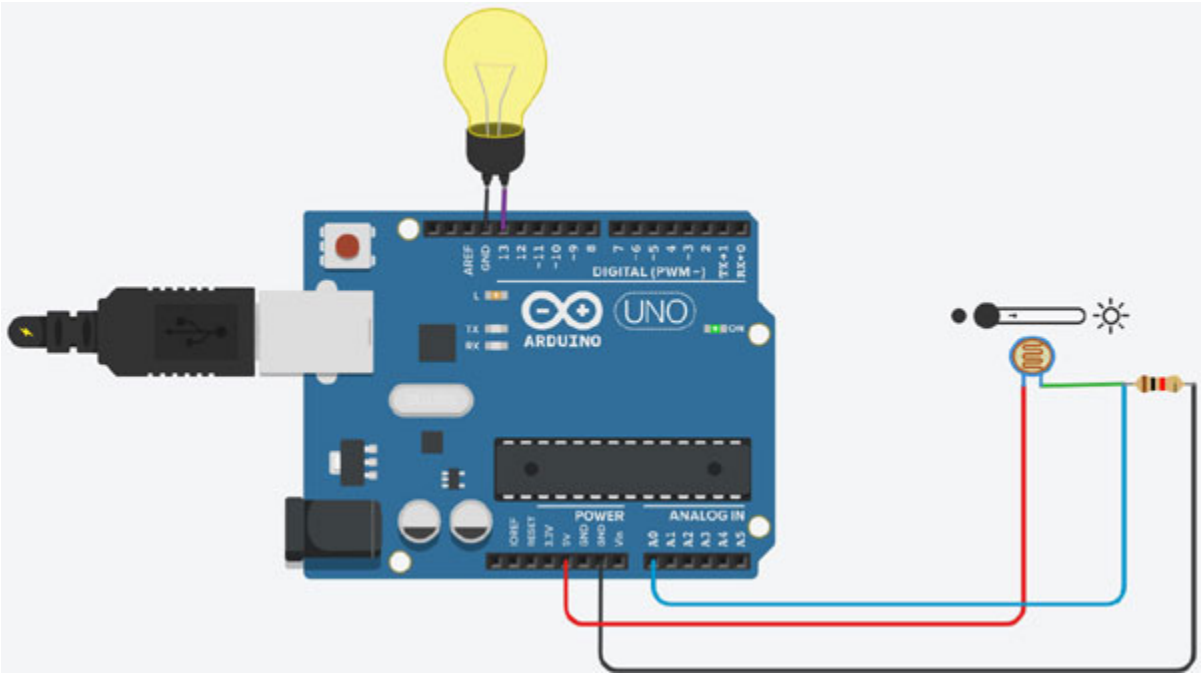


Figure 5.2: *An automatic streetlight system using an LDR with an Arduino board*

The sketch for the recipe is as follows:

```
int light = 0;
void setup()
{
  pinMode(A0, INPUT);
  pinMode(13, OUTPUT);
}
void loop()
{
  light = analogRead(A0);
  if (light < 200) {
    digitalWrite(13, HIGH);
    delay(1000); // Wait for 1000 millisecond(s)
  } else {
    digitalWrite(13, LOW);
    delay(1000); // Wait for 1000 millisecond(s)
  }
}
```

Recipe 2: Detecting the object with an IR proximity sensor

In this recipe, we will discuss the method to detect the presence of an object. The object can be detected using different sensors, and we will discuss a few in this chapter. The first one in this category is an **Infrared** (**IR**) proximity sensor. An IR proximity sensor is designed as a set of IR transmitter and receiver.

Figure 5.3 shows the IR proximity sensor. We can observe that the set of transmitter and receiver is attached to a small PCB. There are only three pins attached to the PCB. One pin is for power, one for ground, and the out pin gives the sensor output. The working principle is also illustrated in the same figure. The transmitter sends waves at 800 Hz. These waves do not interfere with the light waves as they are not in the same spectrum. The receiver is configured to receive the same frequency. If the intensity of the received waveforms is more than a certain threshold, then the output of the sensor is HIGH; else, it is low. The intensity of the incoming waveforms decreases with distance. Therefore, the sensor can detect objects only up to certain distances. The intensity can be changed with the help of a blue-colored potentiometer mounted on the PCB.

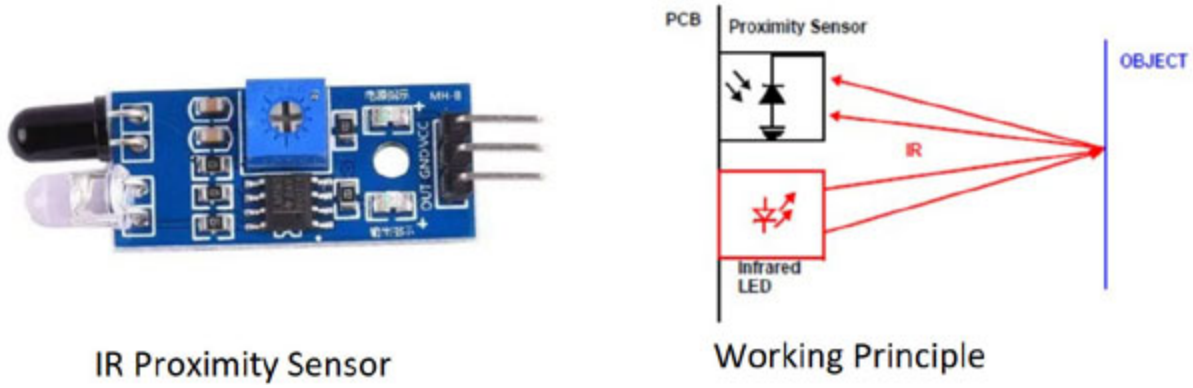


Figure 5.3: *An IR proximity sensor and its working principle*

Let us design the circuit for our recipe now. In this recipe, we have connected the output of the IR proximity sensor to digital pin 4 of Arduino. The connections are shown in *Figure 5.4*. The other two pins of the sensor are connected to the ground and 5V. An LED is attached to PIN 13. The

LED will glow if the output of the sensor is HIGH; else, the LED will remain switched off.

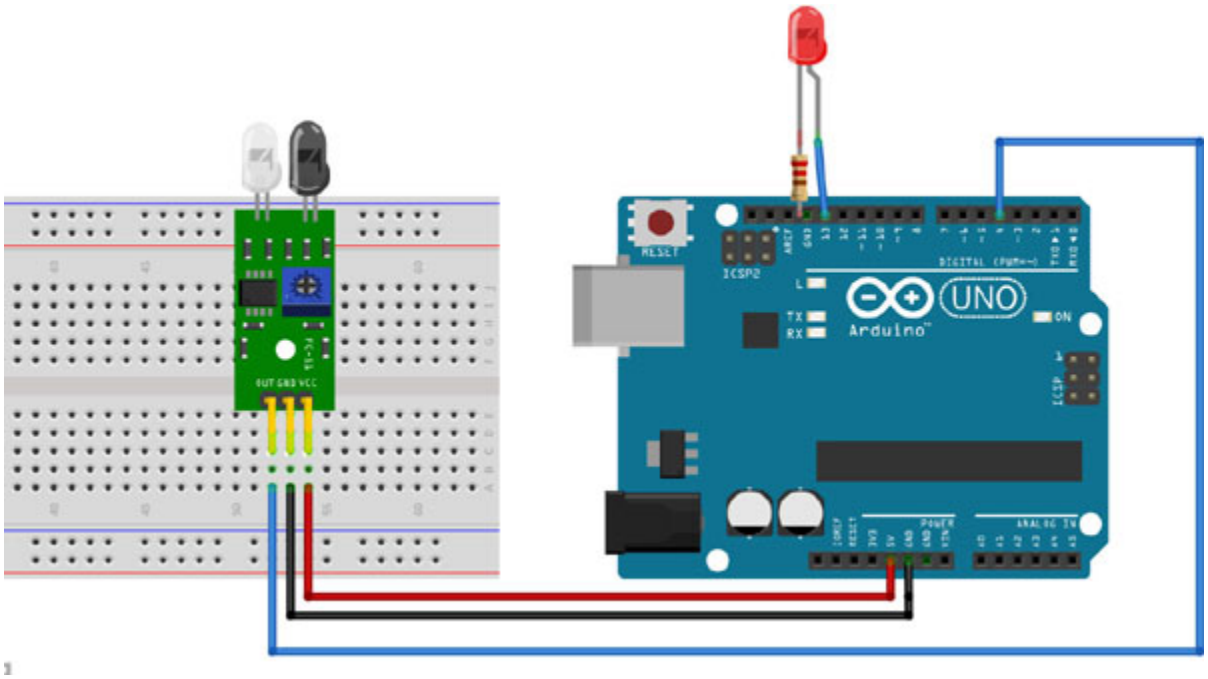


Figure 5.4: *Circuit diagram for detecting an object with an IR proximity sensor*

The sketch of the recipe is as follows:

```
int sensorOut = HIGH;
void setup() {
  pinMode(13, OUTPUT);
  pinMode(4, INPUT);
}
void loop() {
  sensorOut = digitalRead(4);
  if (sensorOut == HIGH)
  {
    digitalWrite(13, HIGH);
    }
    else
    {
    digitalWrite(13, LOW);
    }
```

```
  delay(500);
}
```

Recipe 3: Measuring the distance of a wall from the sensor

The IR proximity sensor is quite simple in its operation. We also observed that it is quite efficient in detecting the presence/absence of an object up to a certain distance. Now, we aim to have better observations. We also want to measure the distance of the detected object from the sensor. IR proximity sensor is not capable of providing these values.

We use an ultrasonic sensor to accurately measure the distance from the sensor. The ultrasonic sensor is shown in *Figure 5.5*. The sensor has two mesh-protected elements on the front. One of them transmits the ultrasonic sound waves, and the other receives the same. These waves travel at the speed of sound. The waves are transmitted at a regular interval. Therefore, we can calculate the time between transmission and reception when we receive the waves:

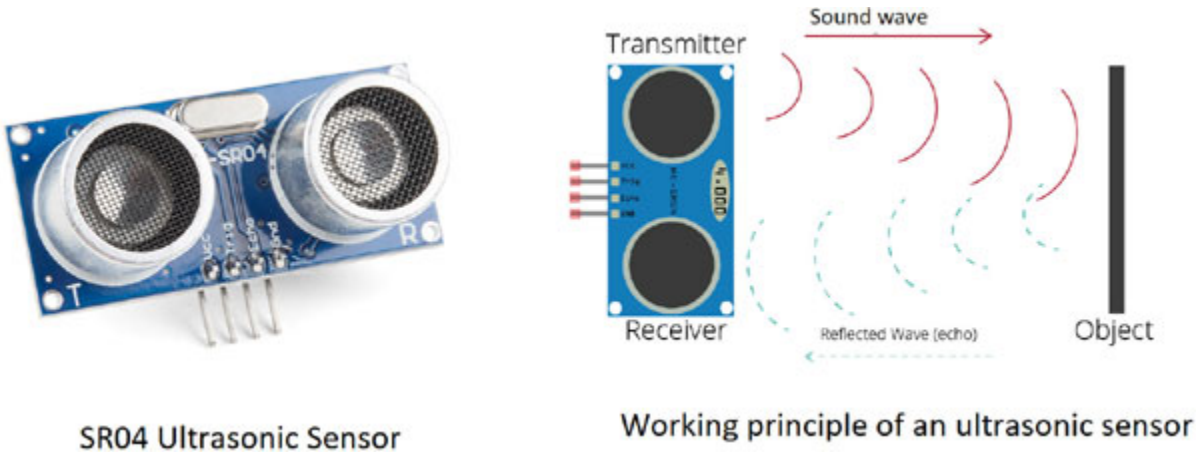


Figure 5.5: *An ultrasonic sensor and its working principle*

The circuit for the recipe is shown in *Figure 5.6*. Let us understand the connections now.

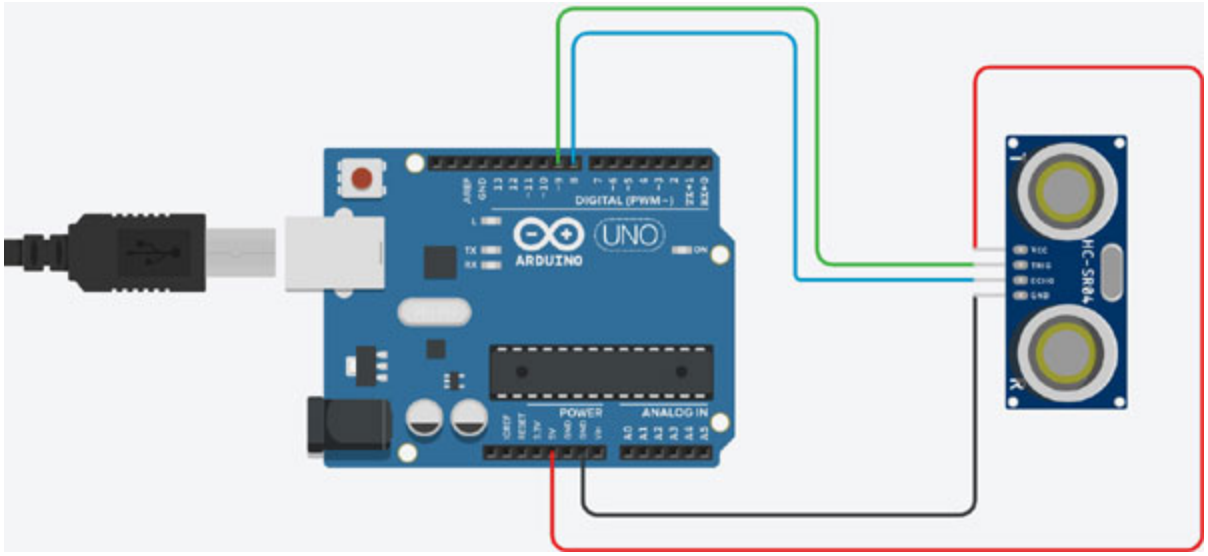


Figure 5.6: *The circuit diagram to connect an ultrasonic sensor to an Arduino board*

In this circuit, we have connected the trigger pin of the ultrasonic sensor to PIN 9 of the Arduino board. The echo pin is connected to PIN 8 of Arduino. The trigger pin is used to transmit the ultrasonic waves, and the eco pin will receive the reflected waves. In the coding, we have used the **`readUltrasonicDistance`** function. This function helps us in calculating the distance based on the trigger and echo pin values. In this function, we are initially sending a LOW signal from the trigger pin, and after a gap of 2 microseconds, we are sending a HIGH signal. After a gap of 10 microseconds, we are sending a LOW signal again. Then, we read the value at the **`echo`** pin. We repeat this process in the **`readUltrasonicDistance`** function. Sound waves travel at a speed of 334 m/s. The distance covered by these waveforms is double the distance of the object from the sensor because the wave is first transmitted to the object and then received back from the object by the sensor. Therefore, we multiply the output of the **`readUltrasonicDistance`** function with 0.01723 to get the distance value in centimeters. We use the serial monitor to display the value of the observed distance:

```
int Distance = 0;
long readUltrasonicDistance(int triggerPin, int echoPin)
{
  pinMode(triggerPin, OUTPUT); // Clear the trigger
  digitalWrite(triggerPin, LOW);
```

```
  delayMicroseconds(2);
  digitalWrite(triggerPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(triggerPin, LOW);
  pinMode(echoPin, INPUT);
  return pulseIn(echoPin, HIGH);
}
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  Distance = 0.01723 * readUltrasonicDistance(9, 8);
  Serial.print("The observed distance is = ");
  Serial.println(Distance);
  delay(1000); // Wait for 1000 millisecond(s)
}
```

Recipe 4: Clap-controlled home appliance

We are living in a modern world now. Controlling our home appliances, curtains, lights, and gates through voice commands is becoming quite common now. Smart systems like Alexa, Siri and Google Assistant are commonly used nowadays. Do you ever wonder how all of these technologies work? Well, one common feature in all these systems is the voice input. In this recipe, we will design the simplest circuit that can take sounds as input. A microphone is designed to convert sound energy into electrical signals.

Figure 5.7 shows the microphone module that can be used with an Arduino board. The same figure also shows the connection of the microphone module with the Arduino board:

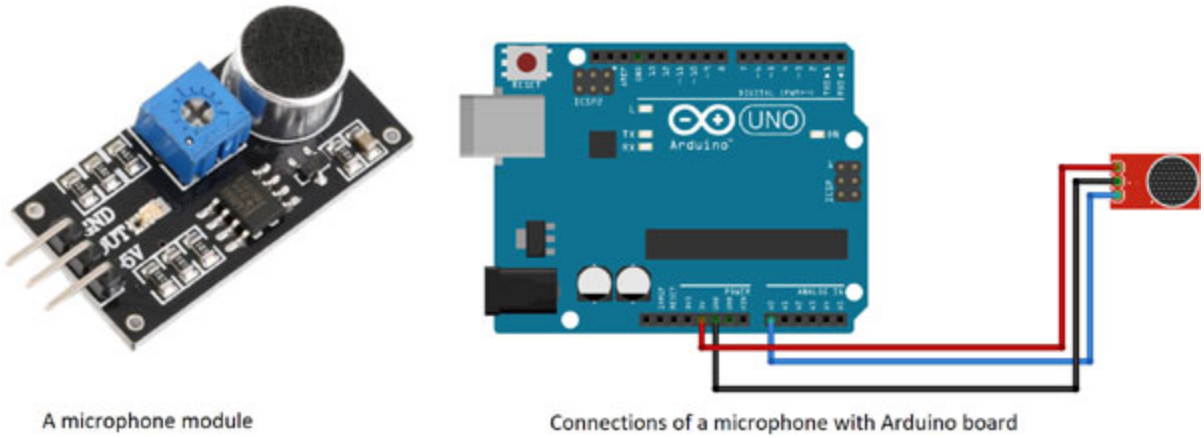


Figure 5.7: *A microphone module and its connections with an Arduino board*

A microphone converts the sound energy into electrical voltage. The microphone output is an analogue signal, which is why the microphone module output pin should be connected to one of the analogue pins of the Arduino board. In this recipe, we have connected the output of the microphone module to pin A0. We connected the microphone module and checked the values on the serial monitor. Now, we should clap or scream and check the values we are getting on the serial monitor. In this recipe, we observed that on clapping, the integer values available at PIN A0 were greater than 600. In the programming part, we use the if-else statement and control an LED at PIN 13 whenever a clap is detected. The sketch for the recipe is as follows:

```
int sound_level = 0;
void setup()
{
  Serial.begin(9600);
  pinMode(13, OUTPUT);
  pinMode(A0, INPUT);
}
void loop()
{
  sound_level = analogRead(A0);
  if(sound_level > 600){
     digitalWrite(13, HIGH);
   }
```

```
    else
    {
     digitalWrite(13, LOW);
    }
    Serial.println(sound_level);
    delay(1000);
}
```

Recipe 5: Designing a water level indicator by measuring the depth of the water level

In the previous chapter, we designed a water level indicator using metal contacts submerged in the water tank. Now that we know about a few different sensors, let us design the same water level indicator system using an ultrasonic sensor. In this recipe, we will use an ultrasonic sensor mounted on the top of our water tank, pointing downward. The ultrasonic sensor will measure the distance of water from the top of the tank. The distance will be the highest when the tank is empty, and it will be the lowest when the tank is full. The circuit for the recipe is shown in *Figure 5.8*:

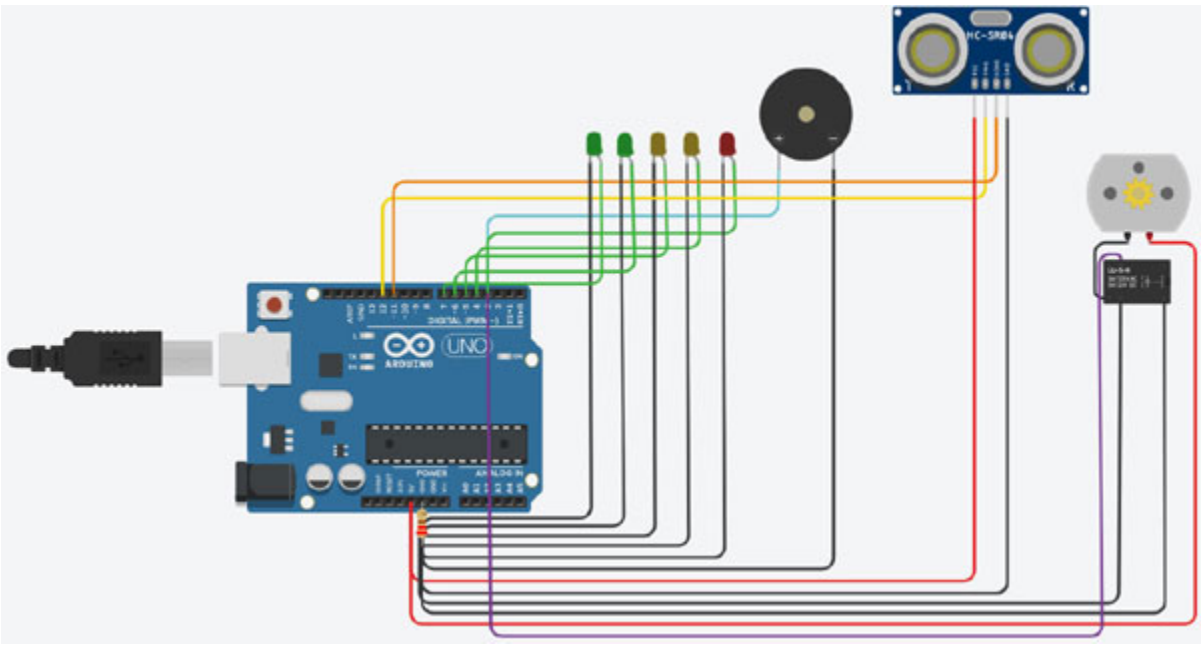


Figure 5.8: *The circuit diagram for an ultrasonic sensor-based water level indicator*

In this circuit, we attached an ultrasonic sensor and used 5 different leads to show different water levels in the tank. We have also added the functionality of controlling the water pump in the same recipe. A relay is connected to one of the output pins, and we will start the water pump whenever we get a minimum water level. The water pump is switched off when a full tank is detected. Let us understand the code for the recipe in parts now. In the setup function, we set the pin modes of all the pins as output and start the serial communication:

```
void setup()
{
  pinMode(7, OUTPUT);
  pinMode(6, OUTPUT);
  pinMode(5, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(3, OUTPUT);
  Serial.begin(9600);
}
```

The **`readUltrasonicDistance()`** function is used to calculate the distance from the input and output of the ultrasonic sensor. We send the trigger from pin number 12 as high and low values at regular intervals and receive them at pin number 11:

```
long readUltrasonicDistance(int triggerPin, int echoPin)
{
  pinMode(12, OUTPUT); // Clear the trigger
  digitalWrite(12, LOW);
  delayMicroseconds(2);
  // Sets the trigger pin to HIGH state for 10 microseconds
  digitalWrite(12, HIGH);
  delayMicroseconds(10);
  digitalWrite(12, LOW);
  pinMode(11, INPUT);
  // Reads the echo pin, and returns the sound wave travel time
  in microseconds
  return pulseIn(11, HIGH);
```

```
}
```

The **`loop()`** function is the main function for this recipe. Initially, we calculate the distance in centimeters and store it in the variable level. Then, we apply different if-else conditions to decide the number of LEDs that will be on or off for a given level of water:

```
void loop()
{
  int level = 0.01723 * readUltrasonicDistance(12,11);
  Serial.print(“ The water level is “);
  Serial.println(level);
  if( level > 99){
   digitalWrite(7,HIGH);
   digitalWrite(6,LOW);
   digitalWrite(5,LOW);
   digitalWrite(4,LOW);
   digitalWrite(3,LOW);
   Serial.println(“Tank is almost empty”);
   delay(100);
  }
  else if ( level > 80){
   digitalWrite(7,HIGH);
   digitalWrite(6,HIGH);
   digitalWrite(5,LOW);
   digitalWrite(4,LOW);
   digitalWrite(3,LOW);
   Serial.println(“Tank is 20 percent full”);
   delay(100);
  }
  else if (level > 60){
   digitalWrite(7,HIGH);
   digitalWrite(6,HIGH);
   digitalWrite(5,HIGH);
   digitalWrite(4,LOW);
   digitalWrite(3,LOW);
```

```
    Serial.println("Tank is 50ercent full");
    delay(100);
  }
  else if (level > 40){
    digitalWrite(7,HIGH);
    digitalWrite(6,HIGH);
    digitalWrite(5,HIGH);
    digitalWrite(4,HIGH);
    digitalWrite(3,LOW);
       Serial.println("Tank is 70 percent full");
    delay(100);
  }
  else if (level > 30){
    digitalWrite(7,HIGH);
    digitalWrite(6,HIGH);
    digitalWrite(5,HIGH);
    digitalWrite(4,HIGH);
    digitalWrite(3,HIGH);
       Serial.println("Tank is alost full");
    delay(100);
  }
  else
    digitalWrite(7,LOW);
    digitalWrite(6,LOW);
    digitalWrite(5,LOW);
    digitalWrite(4,LOW);
    digitalWrite(3,LOW);
       Serial.println("No information about level");
    delay(100);
}
```

Recipe 6: Detecting the LPG leakage in a home

In this recipe, we are going to talk about a new sensor: a gas sensor. Our environment has a mixture of different gases at certain moments. The

density of a certain gas can become more than normal. For example, if we go out in a polluted environment, carbon dioxide or carbon monoxide can be in higher quantity than usual. Similarly, if there is a gas leakage in an industrial plant, we can observe that the density of that leaked gas is more than the usual levels. We have different gas sensors to detect the density of any gas.

We use the MQ series of gas sensors to detect different environmental gases. Some of the sensors designed for respective gases are mentioned in *Table 5.1*:

Sensor name	Gases to detect	Sensor name	Gases to detect
MQ-2	Methane, Butane, LPG, smoke	MQ-6	LPG, butane gas
MQ-3	Alcohol, Ethanol, smoke	MQ-7	Carbon Monoxide
MQ-4	Methane, CNG Gas	MQ-8	Hydrogen Gas
MQ-5	Natural gas, LPG	MQ-9	Carbon Monoxide, flammable gasses

Table 5.1: *Different gas sensors in the MQ series of gas sensors*

There are a few more sensors for different other gases. The working principle and the physical structure of all these sensors are very similar. The chemical configuration inside a sensor is changed based on the gas it detects. All these sensors come in PCB form, and they have 6 connecting pins. In this recipe, we are using the sensor MQ 6 for detecting the LPG. A gas sensor can provide 2 types of outputs: analogue and digital. The digital output just informs us that there is a presence or absence of the particular gas, whereas the analog output informs us about the density of the gas. This recipe uses the analog output and compares the detected gas level to trigger an alarm.

The circuit diagram for the recipe is shown in *Figure 5.9*. The input from the gas sensor is taken at PIN A0, and the output for the alarm is from digital pin 4.

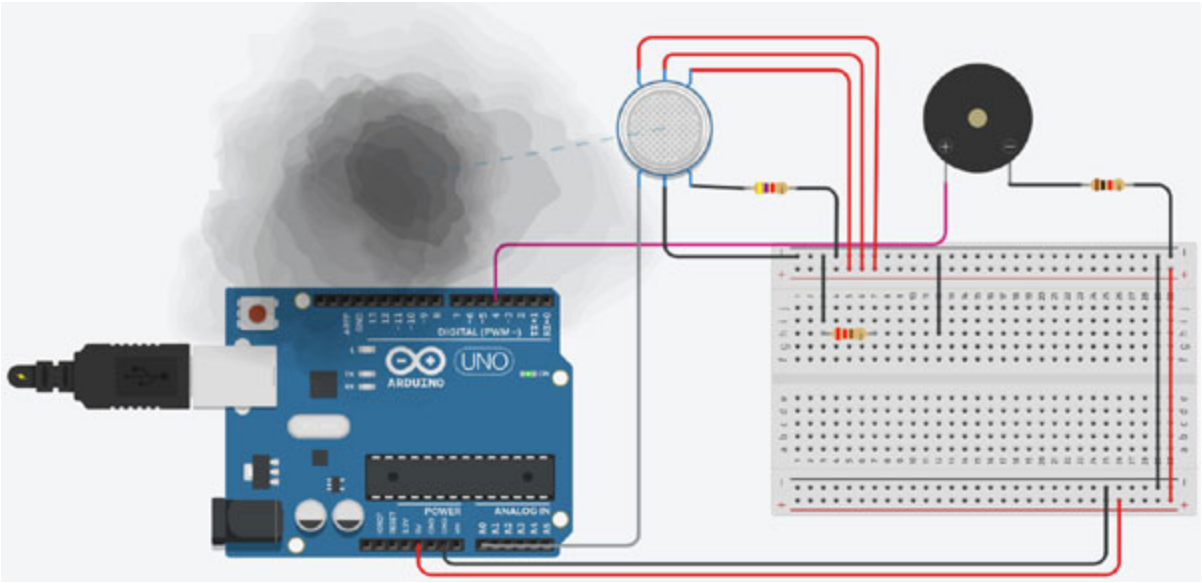


Figure 5.9: *The circuit to detect the presence of LPG in the environment*

The sketch for the recipe is simple. We will note down the values of gas density from the data of PIN A0 and use a simple if-else condition to control the output at PIN 4. The sketch of the recipe is as follows:

```
int buzzer = 4;
int sensor = A0;
int sensorThresh = 400;
void setup()
{
pinMode(buzzer,OUTPUT);
pinMode(sensor,INPUT);
Serial.begin(9600);
}
void loop()
{
 int analogValue = analogRead(sensor);
 Serial.println(analogValue);
 if(analogValue>sensorThresh)
 {
  digitalWrite(buzzer,HIGH);
  delay(1000);
 }
```

```
else
{
 digitalWrite(buzzer,LOW);
 delay(1000);
}
}
```

Recipe 7: Detect the motion of a person around your home

In this recipe, we are going to discuss a new sensor: a **Passive Infrared Sensor** (**PIR**). It is used to detect the movement of any heat source. The sensor has a Fresnel lens on the front, and it detects movement in front of the lens. This sensor is digital in nature, which means it will only tell us whether there is a movement.

Figure 5.10 shows the PIR sensor connected with an Arduino board. A PIR sensor has 3 pins: two are used for power and ground, and the third is the output pin. The output pin of the P IR sensor is connected to digital pin 2 on the Arduino board. We have connected a buzzer at PIN 12. The buzzer will raise the alarm whenever any movement is detected in front of the PIR sensor.

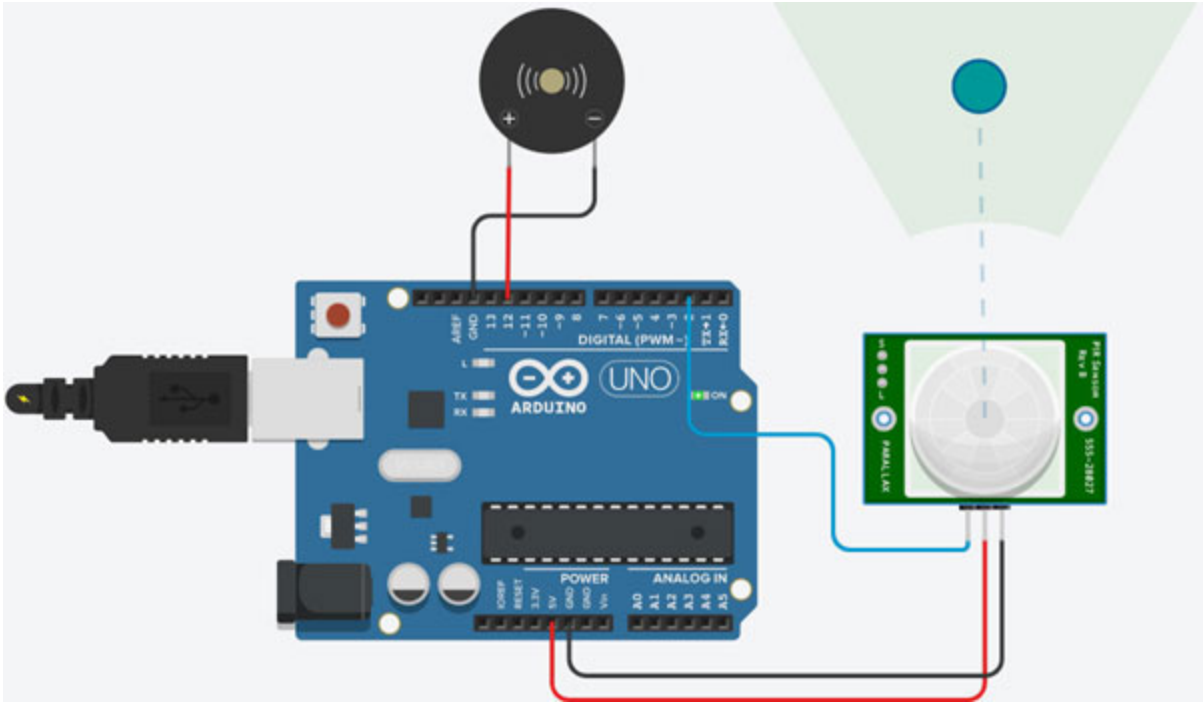


Figure 5.10: *A PIR sensor connected to an Arduino to detect any movement*

The coding for this recipe is very simple. We read the digital input at PIN 2 and control the digital output at PIN 12. Whenever the digital input 8P number 2 is HIGH, we raise the alarm at PIN 12. The sketch for the recipe is as follows:

```
void setup()
{
  pinMode(2, INPUT);
  pinMode(12, OUTPUT);
}
void loop()
{
  if (digitalRead(2) == 1) {
    digitalWrite(12, HIGH);
    delay(3000); // Wait for 3000 millisecond(s)
  } else {
    digitalWrite(12, LOW);
  }
}
```

Recipe 8: Design a robust burglar alarm system

In the previous recipe, we designed a circuit to detect any heat source's movement. This circuit is good for beginners, but it is not quite good for actual implementation. The problem with the previous recipe is that the system will raise the alarm if there is any movement, even by a tree, an animal, or any person around. Now we want to design a burglar alarm system that will raise an alarm only when some suspicious activities are detected on the premises. We will change the previous circuit and design a robust burglar alarm system.

The circuit diagram for the robust burglar alarm is shown in *Figure 5.11*. The main problem in the previous system was that it could detect any movement. In this recipe, we have set the sensitivity of the PIR sensor using a potentiometer. When the alarm is raised, we have added an extra pushbutton to turn it off. We have added an additional function to stop the alarm using the interrupt pins.

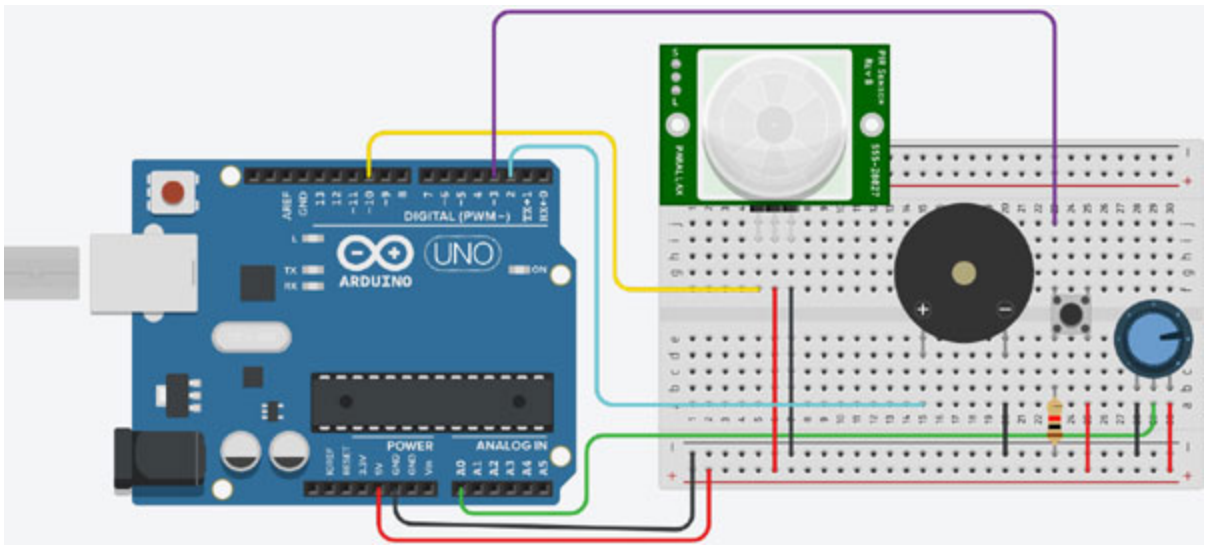


Figure 5.11: *A burglar alarm circuit using the PIR sensor*

We have made appropriate changes in the code as well. We have declared additional variables like **`BtnStatus`** for knowing the status of button voltage, **`SensorStatus`** to know if the sensor is detecting a movement, **`AlarmStatus`** to know whether the alarm is ringing, **`AlarmSetterCounter`** to count the number of movements, and **`AlarmSensitivity`** to set the sensitivity of the alarm. We have also used the serial monitor to regularly

check the status of all these variables. First, only the movements that are more than the threshold set by the potentiometer are considered. Even then, even if the sensor detects a movement, the alarm is not raised immediately. We count for 20 such movements and then raise the alarm. The alarm can be switched off using the pushbutton.

The sketch for the recipe little lengthy; let us understand it in parts. Initially, we declare the required variables. The important ones are related to alarm, interrupt pins, sensor value, potentiometer pin, button status and sensor status:

```
const int Alarm = 2;
const byte interruptPin = 3;
const int Sensor = 10;
const int Pot = A0;
int BtnStatus = 0;
int SensorStatus = 0;
bool AlarmStatus= false;
int AlarmSetterCounter=0;
int AlarmSensitivity=20;
```

In the **`setup()`** function, we define the **`pinModes`** of different pins and start the serial communication:

```
void setup()
{
   pinMode(Alarm, OUTPUT);
   pinMode(Sensor, INPUT);
   pinMode(Pot, INPUT);
   attachInterrupt(digitalPinToInterrupt(3),stop,CHANGE);
  Serial.begin(9600);
}
```

The **`loop()`** function is the main function of this recipe. In this function, we set the alarm sensitivity to 20; this means we will trigger the alarm only if movement is detected 20 times. Based on the status of alarm, the alarm is triggered. If the alarm needs to be stopped in between, the **`stop()`** function is called:

```
void loop()
```

```
{
 SensorStatus = digitalRead(Sensor);
 AlarmSensitivity = 20*((float)analogRead(Pot)/(float)1024);
 if (SensorStatus == HIGH)
 {
  AlarmSetterCounter++;
  Serial.println(AlarmSetterCounter);
  if(AlarmSetterCounter>=AlarmSensitivity)
  {
   AlarmStatus=true;
  }
  else
  {
   AlarmStatus=false;
  }
 }
 else
 {
  AlarmSetterCounter=0;
  digitalWrite(Alarm,LOW);
 }
 if(AlarmStatus)
 {
  digitalWrite(Alarm,HIGH);
  Serial.println(“detected”);
  delay(10000);
  Serial.println(SensorStatus);
  Serial.println(AlarmSetterCounter);
  AlarmSetterCounter=0;
  AlarmStatus=false;
 }
 else
 {
  digitalWrite(Alarm,LOW);
```

```
    delay(100);
    AlarmStatus=false;
  }
}
```

The stop() function is used to stop the buxxing alarm:

```
void stop()
{
  digitalWrite(Alarm,LOW);
  AlarmSetterCounter=0;
     Serial.println("triggered");
     Serial.println(AlarmSetterCounter);
  delay(1000);
  }
```

Recipe 9: Securing access to a door using a 4-digit password

We must have seen some doors where we must enter a password to unlock the door. These kinds of doors have a keypad on the panel, and we must press the password keys. If we enter the correct password, the door unlocks, and we are given access through the door. In this recipe, we will design a similar system, allowing a person to pass through a door. In this recipe, we are going to talk about a new input device: a keypad. Generally, a 4x4 keypad is used to provide the inputs to the microcontrollers.

A commonly available 4x4 keypad has 16 keys arranged in 4 rows and 4 columns. *Figure 5.12* shows one such keypad, along with its internal circuit connections. The circuit for a 4x4 keypad is quite simple.

Every key is a pushbutton. Whenever a key is pressed, the signals for the corresponding row and column numbers are set to HIGH. Therefore, we can get a unique 8-bit code (made by a combination of 4 rows and 4 columns) for every key pressed. We use the **`keypad.h`** library, which can map these 8-bit keys to the respective characters printed on the top of every key. Refer to the following figure:

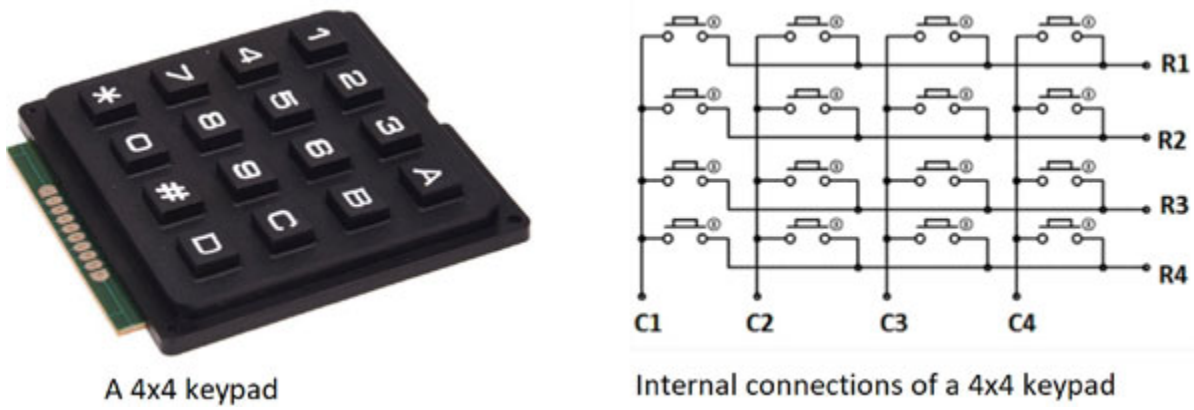


Figure 5.12: *A 4x4 keypad and its internal connections*

For coding, we use the **`keypad.h`** library to include all the functions related to the keypad. There are various types of keypads supported by the library. Therefore, we must define the configuration of our keypad. We use the following array declaration to map the row and column characters. Refer to *Figure 5.12* and the character values in the following declaration. If you have another type of keypad, then we must use this array accordingly:

```
const byte COLS = 4; //four columns
char keys[ROWS][COLS] = {
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
```

The circuit connections for the recipe are shown in *Figure 5.13*. We have connected 4 rows and 4 columns of the keypad to 8 different pins of the Arduino board.

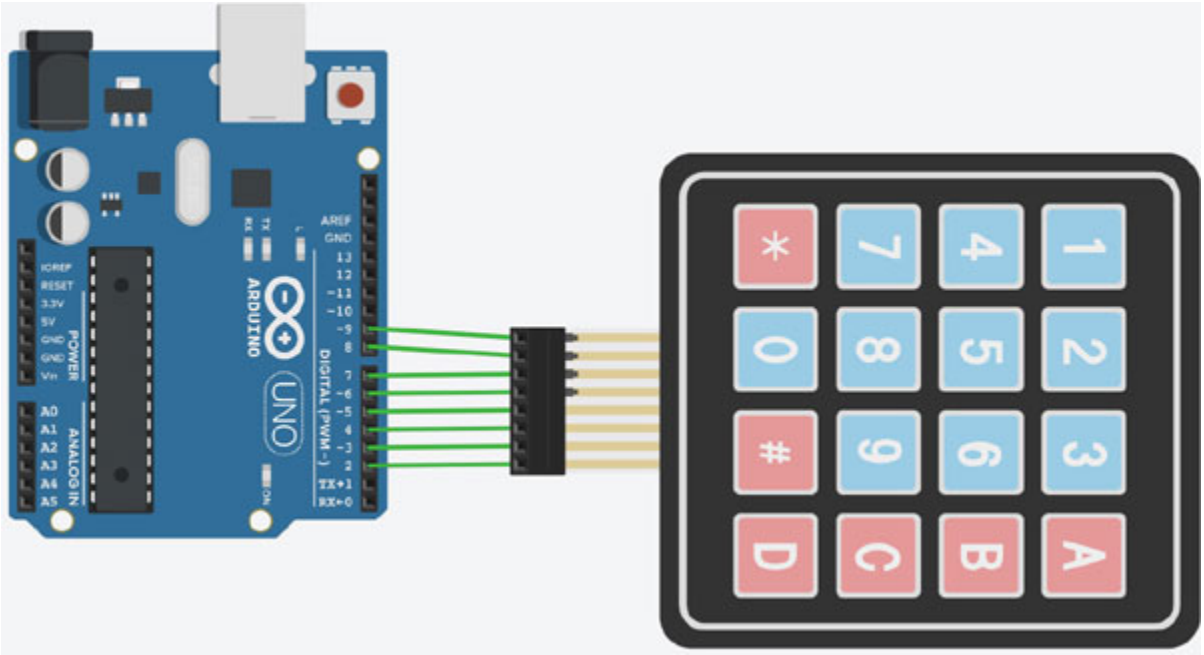


Figure 5.13: *Connections of a 4x4 keypad with an Arduino board*

The sketch for the recipe is as follows:

```
#include <Keypad.h>
const byte ROWS = 4; //four rows
const byte COLS = 4; //four columns
char keys[ROWS][COLS] = {
  {‘1’,’2’,’3’,’A’},
  {‘4’,’5’,’6’,’B’},
  {‘7’,’8’,’9’,’C’},
  {‘*’,’0’,’#’,’D’}
};
byte rowPins[ROWS] = {9, 8, 7, 6}; //connect to the row
pinouts of the keypad
byte colPins[COLS] = {5, 4, 3, 2}; //connect to the column
pinouts of the keypad
Keypad keypad = Keypad( makeKeymap(keys), rowPins, colPins,
ROWS, COLS );
String v_passcode=””;
void setup()
{
```

```
 Serial.begin(9600);
}
void loop(){
 char key = keypad.getKey();
 if (key != NO_KEY){
  //Serial.println(key);
  v_passcode = v_passcode + key;
  if(key==’*’)
  {
   Serial.println(“Enter Password ending with D”);
   v_passcode=””;
  }
  if(key==’#’)
  {
   Serial.println(“Enter Password again ending with D”);
   v_passcode=””;
    Serial.println(v_passcode);
  }
  if(key==’D’)
  {
   Serial.println(“Validating the Password”);
   Serial.println(v_passcode);
    if (v_passcode==”589D”)
    {
     Serial.println(“Access Granted”);
     v_passcode=””;
    }
    else
 {
       Serial.println(“Access Denied”);
        v_passcode=””;
   }
  }
 }
```

```
}
```

Recipe 10: Designing a multi-password door lock

In the previous recipe, we designed a system to unlock the door by entering the correct password. Multiple people are given different passwords in a typical office to unlock a door. So in this recipe, we will design an extended version of the previous recipe and make a multi-password-protected door lock. The recipe's concept is almost identical, and the circuit connections will require a few amendments.

Figure 5.14 shows that there are two bulbs that indicate the excess to two different types of people. In the same figure, we can also observe the output on the serial monitor and whether the access is granted or denied. We have used two bulbs to show whether the access is given. The bulb will glow whenever access is given to a particular person. In the practical system, instead of the bulbs, we will be using servo motors and the related actuators to open the door.

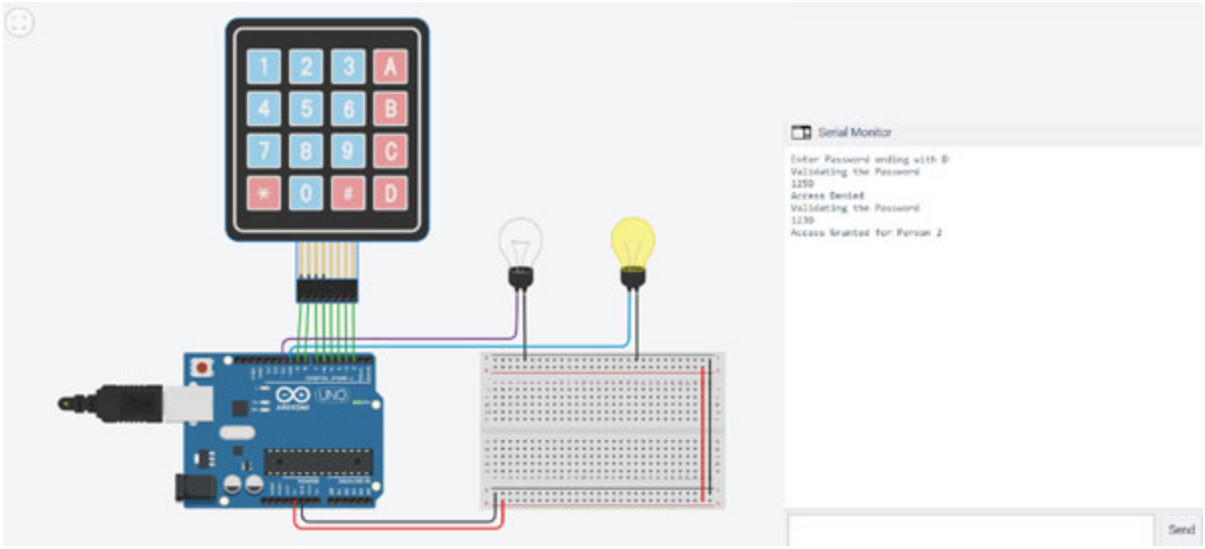


Figure 5.14: *A multi-password protected door*

In the coding part, we introduce additional **`if-else`** statements to accommodate the additional passwords. We can define multiple passwords with multiple if-else cases. The sketch for the recipe is explained in parts. Initially, we declare the **`Keypad.h`** library and the respective connections to

attach a keypad to the Arduino. We must mention the key characters in a matrix form to define the keypad inputs:

```
#include <Keypad.h>
const byte ROWS = 4; //four rows
const byte COLS = 4; //four columns
char keys[ROWS][COLS] = {
  {‘1’,’2’,’3’,’A’},
  {‘4’,’5’,’6’,’B’},
  {‘7’,’8’,’9’,’C’},
  {‘*’,’0’,’#’,’D’}
};
byte rowPins[ROWS] = {9, 8, 7, 6}; //connect to the row
pinouts of the keypad
byte colPins[COLS] = {5, 4, 3, 2}; //connect to the column
pinouts of the keypad
Keypad keypad = Keypad( makeKeymap(keys), rowPins, colPins,
ROWS, COLS );
String v_passcode=””;
```

In the **`setup()`** function, we start the serial communication:

```
void setup(){
  Serial.begin(9600);
  pinMode(10, OUTPUT);
}
```

The **`loop()`** function is the major function in this recipe. Initially, we read the inputs of the keypad in variable key. We store the passwords in **`v_passcode`**, and the password must have a length of 4 characters and must end with D. Relevant messages are displayed on detecting the correct or wrong passwords:

```
void loop(){
  char key = keypad.getKey();
  if (key != NO_KEY)
  {
    v_passcode = v_passcode + key;
    if(key==’*’)
```

```
{
   Serial.println(“Enter Password ending with D”);
   v_passcode=””;
}
if(key==’#’)
{
   Serial.println(“Enter Password again ending with D”);
   v_passcode=””;
   Serial.println(v_passcode);
}
if(key==’D’)
{
   Serial.println(“Validating the Password”);
      Serial.println(v_passcode);
      if (v_passcode==”589D”)
      {
      Serial.println(“Access Granted for Person 1”);
       digitalWrite(11,HIGH);
       delay(5000);
       digitalWrite(11,LOW);
       v_passcode=””;
      }
      else if (v_passcode==”123D”)
      {
      Serial.println(“Access Granted for Person 2”);
       digitalWrite(10,HIGH);
       delay(5000);
       digitalWrite(10,LOW);
       v_passcode=””;
      }
      else
   {
       Serial.println(“Access Denied”);
       v_passcode=””;
```

```
      }
    }
  }
}
```

Recipe 11: Designing a reverse car parking assistant system

When we put the car in reverse gear and try to park it, we can hear different sounds inside the car, which help us know the distance of the wall from the car. In today's cars, we even have cameras mounted on the backside to know the exact distance on the dashboard. We will design a simple prototype of a similar car parking assistant system in this recipe. Refer to the following figure:

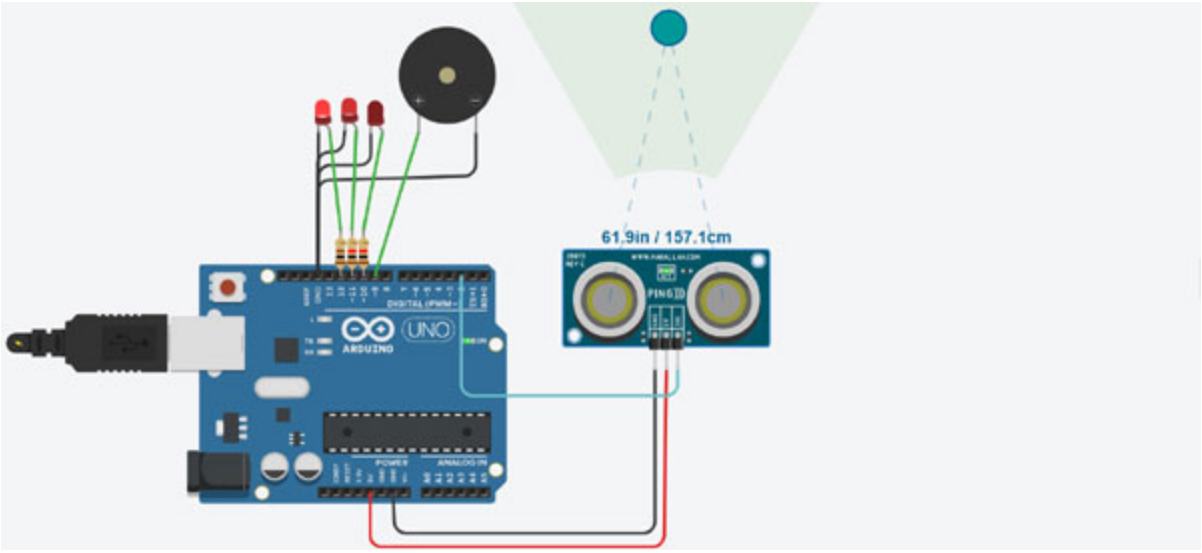


Figure 5.15: *A car parking assistant System designed with an ultrasonic sensor*

Figure 5.15 shows the connections of this system. We have used an ultrasonic sensor to detect and measure the distance of the wall from the rear end of the car. Inside the car, we put multiple LEDs and a buzzer to show the distance of the wolf from the car to the driver. As the distance between the car and the wall reduces, more LEDs will glow. And if the car gets really close to the wall, the alarm will be raised with the help of a buzzer. The sketch for the recipe is as follows:

```
int dist = 0;
```

```
long readUltrasonicDistance(int triggerPin, int echoPin)
{
 pinMode(triggerPin, OUTPUT); // Clear the trigger
 digitalWrite(triggerPin, LOW);
 delayMicroseconds(2);
 digitalWrite(triggerPin, HIGH);
 delayMicroseconds(10);
 digitalWrite(triggerPin, LOW);
 pinMode(echoPin, INPUT);
 return pulseIn(echoPin, HIGH);
}
void setup()
{
 pinMode(9, OUTPUT);
 pinMode(10, OUTPUT);
 pinMode(11, OUTPUT);
 pinMode(12, OUTPUT);
}
void loop()
{
 dist = 0.01723 * readUltrasonicDistance(2, 2);
 if (dist > 200) {
  digitalWrite(9, LOW);
  digitalWrite(10, LOW);
  digitalWrite(11, LOW);
  digitalWrite(12, HIGH);
 } else {
  if (dist > 150 && dist <= 200) {
   digitalWrite(9, LOW);
   digitalWrite(10, LOW);
   digitalWrite(11, HIGH);
   digitalWrite(12, HIGH);
  } else {
   if (dist > 100 && dist <= 150) {
```

```
      digitalWrite(9, LOW);
      digitalWrite(10, HIGH);
      digitalWrite(11, HIGH);
      digitalWrite(12, HIGH);
    } else {
      digitalWrite(9, HIGH);
      digitalWrite(10, HIGH);
      digitalWrite(11, HIGH);
      digitalWrite(12, HIGH);
    }
  }
 }
 delay(10); // Delay a little bit to improve simulation
 performance
}
```

Recipe 12: Measuring the temperature of the surroundings

Temperature is one of the natural phenomena, and we perform many of our daily activities based on this. A high temperature can make us uncomfortable, whereas a pleasant temperature makes our living comfortable in the environment. In factories, we must check and control the temperature regularly.

LM35 is one of the most used temperature sensors. This sensor has three pins. This temperature sensor converts the temperature values into the corresponding voltage signals. For every 1-degree Celsius change in the temperature, there is a change of 10mV in the output voltage.

Figure 5.16 shows the connection of LM35 with the Arduino board. Two pins are connected to the ground and 5V; the output pin is connected to analog pin A4. The temperature sensor provides the analog output, which is why it is connected to the analog pin.

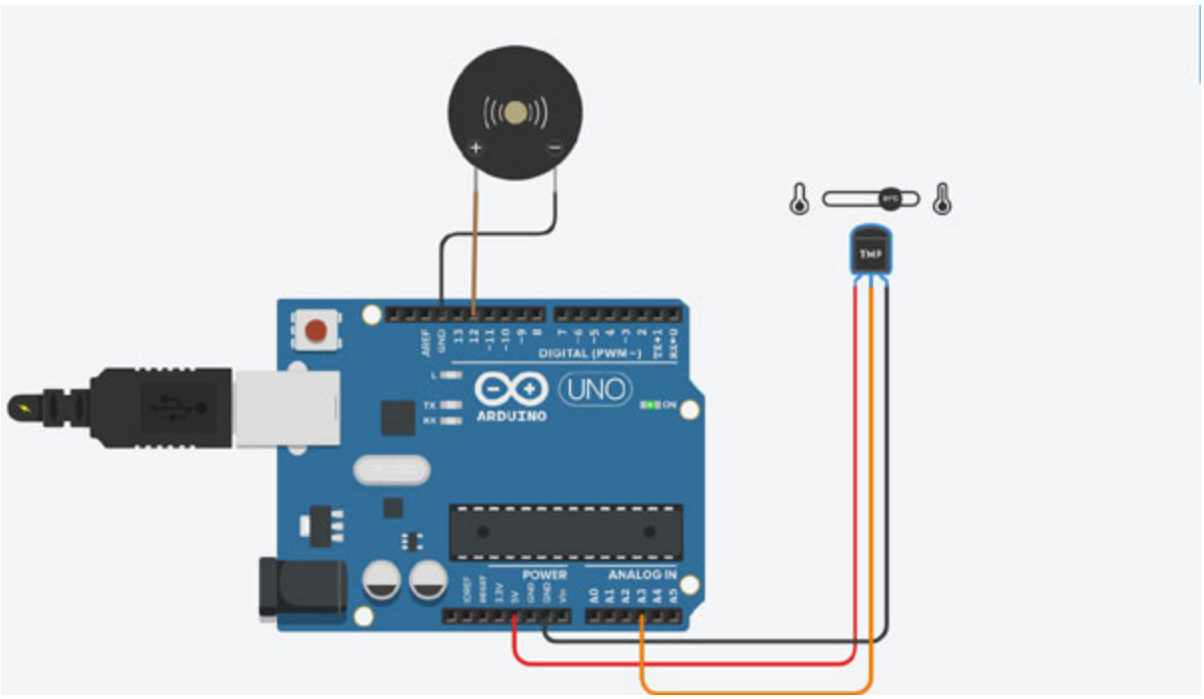


Figure 5.16: *A temperature-controlled alarm using an LM35 sensor*

Every temperature value will generate a unique integer at the analog pin. The coding for this recipe is quite simple. We can decide the threshold voltage for the corresponding threshold temperature beside the output at PIN 12. At PIN 12, we have connected a buzzer to raise the alarm. The sketch for the recipe is as follows:

```
void setup()
{
  pinMode(A3, INPUT);
  pinMode(12, OUTPUT);
}
void loop()
{
  if (analogRead(A3) > 200) {
    digitalWrite(12, HIGH);
    delay(1000); // Wait for 1000 millisecond(s)
  } else {
    digitalWrite(12, LOW);
    delay(1000); // Wait for 1000 millisecond(s)
  }
```

```
}
```

Recipe 13: Designing a digital weather station to show real-time weather conditions

In this chapter, we have discussed various sensors to measure and detect natural phenomenon and their properties. In this recipe, we are going to design a digital weather station. A digital weather station monitors different weather parameters and displays them continuously on a digital display. We can choose the parameters that are relevant to our requirements. In this recipe, we have taken a light sensing LDR, a temperature sensor, a pollution sensor, and a tilt sensor to detect earthquakes. All these sensors continuously sense the natural phenomena associated with them. All these values are simultaneously fed to the Arduino board through different input pins. The Arduino board takes these inputs and displays them on an LCD.

Figure 5.17 shows the circuit diagram for the recipe:

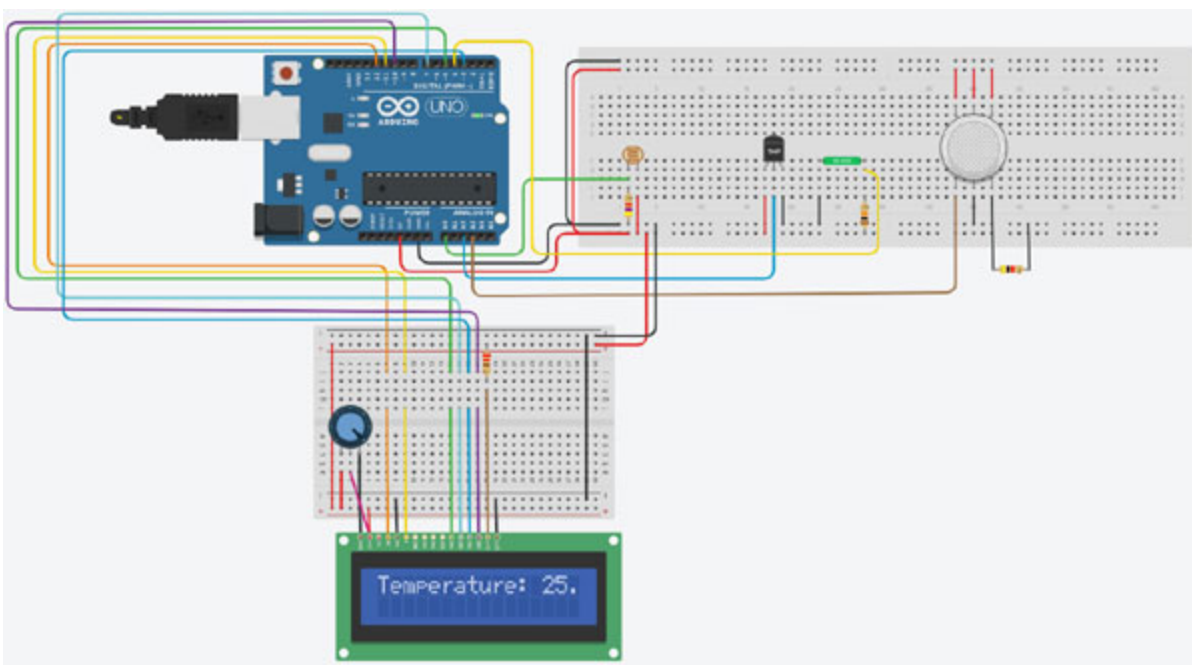


Figure 5.17: *A digital weather station*

In this recipe, we are using a new sensor: a tilt sensor. It is a digital sensor that will indicate whether the surface has tilted more than the permissible limits. We are using this sensor to detect any earthquake around the weather

station. The coding for this recipe is simple. We will continuously display different parameters on the LCD screen, one after another. The sketch for the recipe is a little lengthy; let us understand it is parts. We start by declaring some required variables and libraries. Mainly, LCD library is required, and we mention the pins on which the LCD must be connected:

```
int sensorValue = 0;
int tempVal = A2;
int quakePin = 4;
int quakeState = 0;
int MQ2pin = A3;
const int gas = 0;
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 7, 3, 10);
const int switchPin = 2;
int count = 0;
int switchstate = 0;
```

Now, in the **`setup()`** function, we set up the **`PinModes`** of pins A0 and **`switchPin`**:

```
void setup()
{
  Serial.begin(9600);
  pinMode(A0, INPUT);
  pinMode(switchPin, INPUT);
}
```

In the **`loop()`** function, we call the **`quakeState`** to know whether the earthquake is detected, and we call the **`autoswitcher`** function to display the data of each parameter:

```
void loop()
{
  quakeState = digitalRead(quakePin);
  autoswitcher();
}
```

The **`autoswitcher`** function is the main function of this weather station. We maintain a counter in the function and keep increasing it. For every next

count value, a new function is called to display the information regarding the next weather parameter. Once all the parameters are displayed, the counter is reset to 2 to display the first parameter again:

```
void autoswitcher(){
  lcd.print(“Cycle Mode”);
  delay(1000);
  lcd.clear();
   count++;// add 1 to the count
   if (count >= 2) {
    brightness();
    delay(1000);
    lcd.clear();
    count++;
   }
   if (count >= 3) {
    temp();
    delay(1000);
    lcd.clear();
    count++;
   }
    if (count >= 4) {
    gass();
    delay(1000);
    lcd.clear();
    count++;
   }
   if (count >= 5) {
    quake();
    delay(1000);
    lcd.clear();
    count++;
    if (count >= 6) {
    lcd.clear();
    count = 2;
```

```
    }
  }
}
```

Now, let us discuss about various functions to display respective weather parameters. The brightness function detects the intensity of the light and displays the information about it on LCD. Depending on the various values of light intensity, different messages are also displayed on the LCD:

```
void brightness()
{
  int value = analogRead(A0);
  Serial.println(value);
  if (value <= 600) {
    Serial.println("It's Dark");
      lcd.println("It's Dark");
    }
  if ((value > 600) && (value <= 700)) {
    Serial.println("Comfortable light ");
      lcd.println("Comfortable light");
    }
  if ((value > 700) && (value <= 800)) {
    Serial.println("Noon Brightness");
       lcd.println("Noon Brightness");
  }
  if ((value > 800)) {
    Serial.println("Way too bright");
       lcd.println("Way too bright");
  }
  delay(100);
}
```

The **`temp()`** function monitors the ambient temperature of the place. The temperature value is displayed on the LCD:

```
void temp()
{
  float temp = analogRead(A2); //Read the analog pin
```

```
  temp = (temp * 0.48828125) - 49; // convert output (mv) to
  readable celcius
  Serial.print("Temperature: ");
   lcd.print("Temperature: ");
  Serial.print(temp);
     lcd.print(temp);
  Serial.println("C"); //print the temperature status
     lcd.println("C");
  delay(100);
}
```

The **quake()** function detects the input from tilt sensor. If the tilt is detected, then an alarming message is displayed on the LCD.

```
void quake()
{
  quakeState = digitalRead(quakePin);
  if (quakeState == LOW) {
   Serial.println("Earthquake");
   lcd.print("Earthquake");
  } else {
   Serial.println("Not earthquake");
   lcd.print("Not Earthquake");
  }
  delay(100);
}
```

The **gass()** function motors the pollution level at the station. If the pollution level is more than a certain threshold, the alarm message is also displayed:

```
void gass()
{
  int sensorValue;
  sensorValue = analogRead(A3); // read analog input pin 0
  if(sensorValue >= 400){
   lcd.print(sensorValue);
   lcd.println(" |SMOKE DETECTED");
```

```
    Serial.print(sensorValue);
    Serial.println(“ |SMOKE DETECTED”);
  }
  else{
    lcd.print(“Gas Value = “);
    lcd.println(sensorValue);
    Serial.print(“Gas Value = “);
    Serial.println(sensorValue);
  }
  delay(100);
}
```

Conclusion

In this chapter, we got to know a variety of sensors to sense natural and artificial properties in our surroundings. First of all, we learnt the applications of an LDR to detect the intensity of light falling on its surface. We used IR and ultrasonic sensors for detecting objects and measuring distances. We connected a microphone to control a circuit with the sounds of our claps. We then designed a system to detect different types of gases in the atmosphere. Next, we designed a security system by using a PIR sensor that could detect the movements of a stranger in the home. We also learned to design a password-protected door in the end.

In the next chapter, we will learn the connections of different types of motors with an Arduino board.

Points to remember

- Check if the sensor is digital or analog and connect to appropriate pins.
- Tune the sensors properly and always read the values using a serial monitor initially.
- Work in an environment that is not harmful to the sensors or circuit (humid and hot should be avoided).

- Every sensor should be connected with safety, ensuring that no short-circuit is established.
- Soldering, if required, should be done carefully.

Multiple Choice Questions

1. **How many pins are present in the LDR Sensor?**
    a. 1
    b. 4
    c. 2
    d. 5
2. **Which of the following materials would be used for making an LDR?**
    a. Cadmium sulfide
    b. Pure gold
    c. Iron
    d. Aluminum oxide
3. **On a rainy day, lightning strikes. The flash sound is heard 3 seconds after the flash. How far did the lightning strike from you?**
    a. Around 1 km
    b. Less than 100 meters
    c. Cannot be determined
    d. More than 10 km
4. **LM35 provides**
    a. Temperature values in degrees Celsius
    b. An electrical voltage corresponding to the temperature
    c. Temperature in Fahrenheit
    d. Temperature in Kelvins
5. **Which statement describes a sensor?**

a. A device that senses some environmental parameter
b. It is a type of transducer that converts one form of energy into another
c. It can produce output in the form of electrical pulses, current, or voltage
d. All of the above

6. **Which type of lens is generally used on top of the PIR sensors?**

a. Concave lens
b. Fresnel lens
c. Convex lens
d. High powered lens

7. **A PIR sensor detects the movement of a _____.**

a. Heat source
b. Light source
c. Sound source
d. Pressure source

8. **A 4x3 keypad has 4 rows and 3 columns. How many minimum wires need to be connected to get the data from this keypad?**

a. 8
b. 6
c. 7
d. 10

9. **What is the use of the LDR sensor?**

a. Monitors motion of heat source
b. Monitors air pressure on the surface
c. Monitors light intensity
d. Monitors the heartbeat of a person

10. **Sound waves act like _____.**

a. Electrical charge
b. Pressure waves
c. Current
d. X-ray radiation

Answers

1. **b**
2. **a**
3. **a**
4. **b**
5. **d**
6. **b**
7. **a**
8. **c**
9. **c**
10. **b**

CHAPTER 6

Motor Control Using Arduino

Introduction

Motion is an integral part of a lot of Arduino-based projects. The field of robotics is highly dependent on different moving parts. Motors are one of the primary sources to provide rotatory motion in the systems. Even linear motion can be generated by translating rotatory motion into linear motion. Therefore, learning about different types of motors and their connections with an Arduino board is very important.

In this chapter, we will discuss the different types of motors. We will learn the techniques to connect them and the additional tools required to control their movements. We'll start with a hobby DC motor with which a simple AA battery can operate. We'll learn how to connect such motors with gears and control them with an Arduino board. Some of the motors cannot be driven directly by the Arduino board, and we need to learn how to connect an external power source and motor driver for the same. We will learn the design of different robots using the same. We will also discuss servo and stepper motors for limited motions. These motors are used for precisely calculated motions for specific purposes. We'll design a few applications using the same motors.

Structure

In this chapter, we will discuss the following topics:

Connecting DC hobby motors to an Arduino

- Controlling the direction of a hobby motor
- Driving a 5 Volt DC motor using Arduino with L293D
- Controlling the speed of a motor using PWM
- Driving a 12 Volts DC motor using Arduino

- Designing a 2-wheel drive (2WD) robot
- Designing an obstacle avoider robot with Arduino
- Designing a line following robot to run on a black line
- Controlling the angle of movement of a servo motor
- Controlling the movements of a servo with a potentiometer
- A smart dustbin that opens the lid on detecting a person around it
- Automated parking boom barrier
- Designing a robotic hand to pick a glass of water
- Controlling a stepper motor with Arduino
- Controlling a DC motor with an encoder with Arduino

Objectives

By the end of this chapter, you will be able to do the following:

- Connect and run a DC motor with Arduino board
- Know how to use a motor driver for driving multiple motors
- Set a specific angle of rotation for a servo motor
- Make autonomous robots to perform simple operations
- Know and control the rotations of a stepper motor
- Control an encoder base DC motor

Recipe 1: Connecting DC hobby motors to an Arduino

A **Direct Current** (**DC**) operated motor is one of the most widely used actuators around us. This motor converts electrical energy into rotational energy. A DC motor can run at very low DC voltages and higher voltages. A hobby DC motor typically runs at 3V. It is used in various Arduino projects to demonstrate the rotation mechanics. The motor is quite simple in its external configuration. There are two connecting pins to provide the external power supply. One pin is typically connected to the ground and the other to the supply voltage.

A DC has two terminals: one is positive, and the other one is negative. The connections for running the motor are quite simple. We can also run a DC motor using a simple battery. Connect the positive terminal of the battery to the positive terminal of the motor and the negative terminal of the battery to the negative terminal of the motor, and the motor will rotate.

In this recipe, we will control a DC motor with Arduino. The circuit diagram for the recipe is shown in *Figure 6.1*. The connections are quite simple. We have connected the motor's positive terminal to digital pin 12, and the negative terminal of the motor to the ground pin of Arduino.

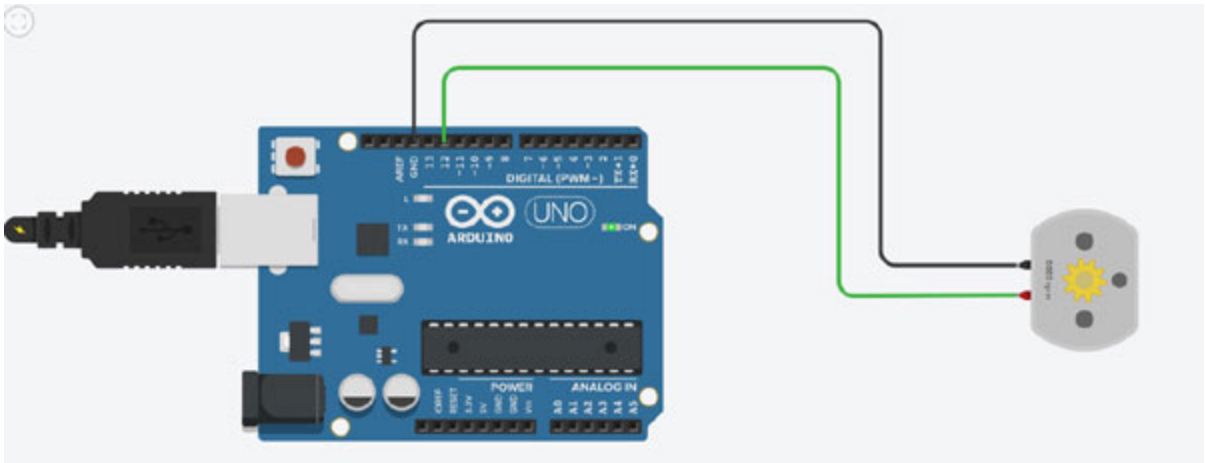


Figure 6.1: *Connecting a DC motor to an Arduino*

The sketch for the recipe is shown as follows. We set the digital pin 12 HIGH for a duration of 5 seconds and low for 1 second. The process is repeated forever:

```
void setup()
{
 pinMode(12, OUTPUT);
}
void loop()
{
 digitalWrite(12, HIGH);
 delay(5000);
 digitalWrite(12, LOW);
 delay(1000);
}
```

Recipe 2: Controlling the direction of a hobby motor

In the previous recipe, we learned how to run and stop a DC motor using an Arduino. In that recipe, we can perform only two simple operations that are either running or stopping the motor. The motor is always rotating in one direction. There are various occasions when we must also rotate the DC motor in the opposite direction. A very simple way of doing it is to change the polarity of connections on the motor terminals. That means if we connect higher potential on the negative terminal and lower potential on the positive terminal, then the motor will rotate in the anticlockwise direction.

This recipe will explore the methods of rotating the DC motor in any direction. Changing the direction of rotation requires a change in the polarity as well. Therefore, we connect both terminals to digital pins in this recipe. Now, we can change the polarity of both the pins in either way to decide the rotation direction of the motor. *Figure 6.2* shows the connections for the recipe:

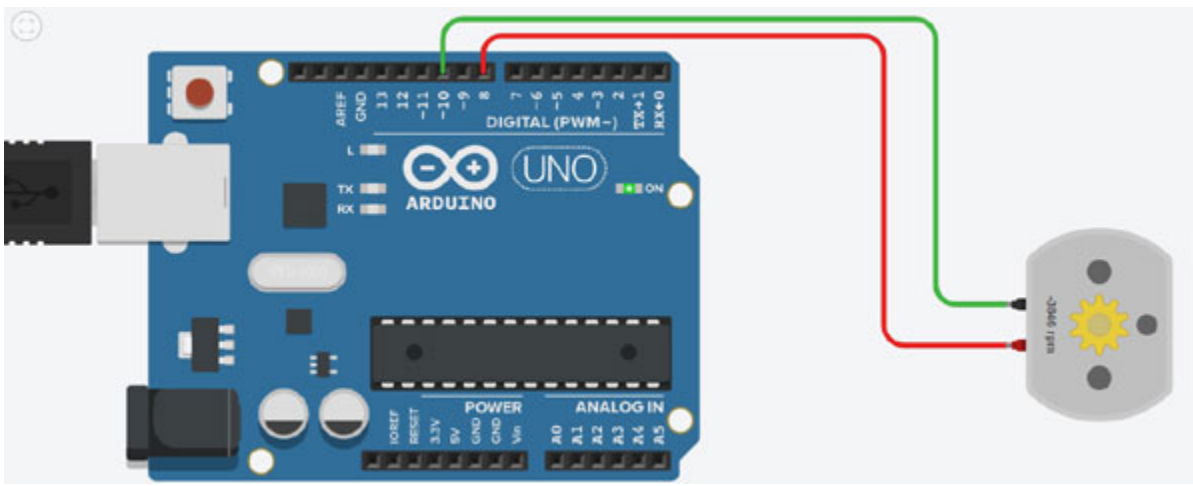


Figure 6.2: *Controlling the direction of rotation of a DC motor*

Using two programmable pins, we can have four different combinations. The sketch for the recipe is given below; let us understand all these four combinations in detail. Pin number eight is the motor's positive terminal, and PIN 10 is the motor's negative terminal. In the first combination, we have provided HIGH voltage at pin 8 and low voltage at pin 10; therefore, the motor rotates in the clockwise direction for two seconds. In the next combination, we have set both the pins to HIGH, and the motor will not

rotate because there is no potential difference between the 2 pins. In the next combination, we have reversed the polarity for both the pins, and the motor rotates anticlockwise for 2 seconds. In the final combination, we again gave the same polarity to both pins, and the motor will not run. The code for the recipe is as follows:

```
void setup()
{
 pinMode(10, OUTPUT);
 pinMode(8, OUTPUT);
}
void loop()
{
 digitalWrite(10, LOW);
 digitalWrite(8, HIGH);
 delay(2000);
 digitalWrite(10,HIGH);
 digitalWrite(8, HIGH);
 delay(2000);
 digitalWrite(10,HIGH);
 digitalWrite(8, LOW);
 delay(2000);
 digitalWrite(10,LOW);
 digitalWrite(8, LOW);
 delay(2000);
}
```

Recipe 3: Driving a 5 Volt DC motor using Arduino with L293D

Controlling a DC motor with Arduino seems to be a very easy task. The same DC motor can be connected with a combination of gears to run different other moving parts. For example, the tires of a robot are generally run by geared motors. These motors run at the same voltage but require more current to generate enough torque to rotate the tires. The maximum

output from Arduino digital pin can be 20mA, and this current is not sufficient to drive geared motors.

We need the motor drivers to drive the DC motor with a higher current requirement. L293D Is a very popular motor driver for driving DC motors through Arduino or any other microcontroller. In this recipe, we will talk about controlling a 5 Volt DC motor using the same motor driver.

Figure 6.3 shows the L293D IC with its pin configuration and the Printed Circuit Board (PCB) packaging available for easy connections.

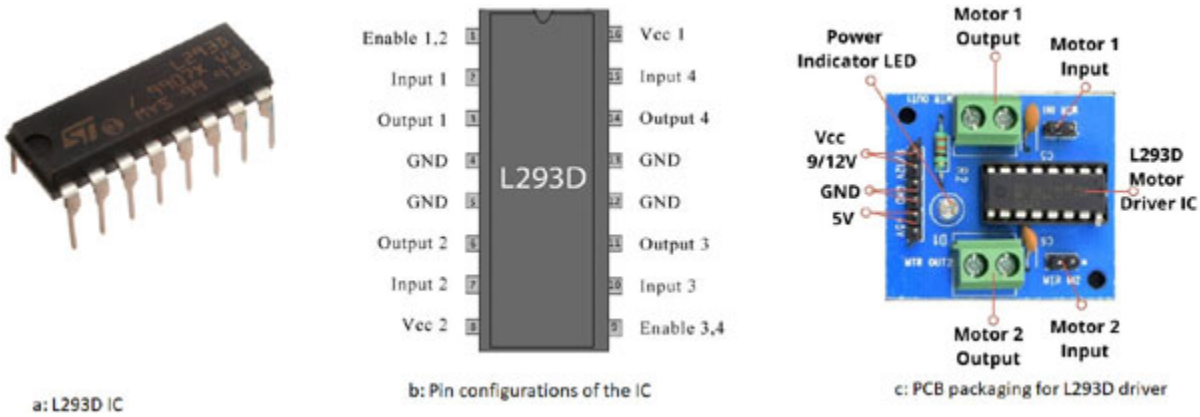


Figure 6.3: *L293D IC and its PCB package available for easy connections*

In this recipe, we will control two different geared motors using a single L293D IC. We will rotate both in a single direction, so we will control the motor with a single programmable pin. The connections for the recipe are shown in *Figure 6.4*:

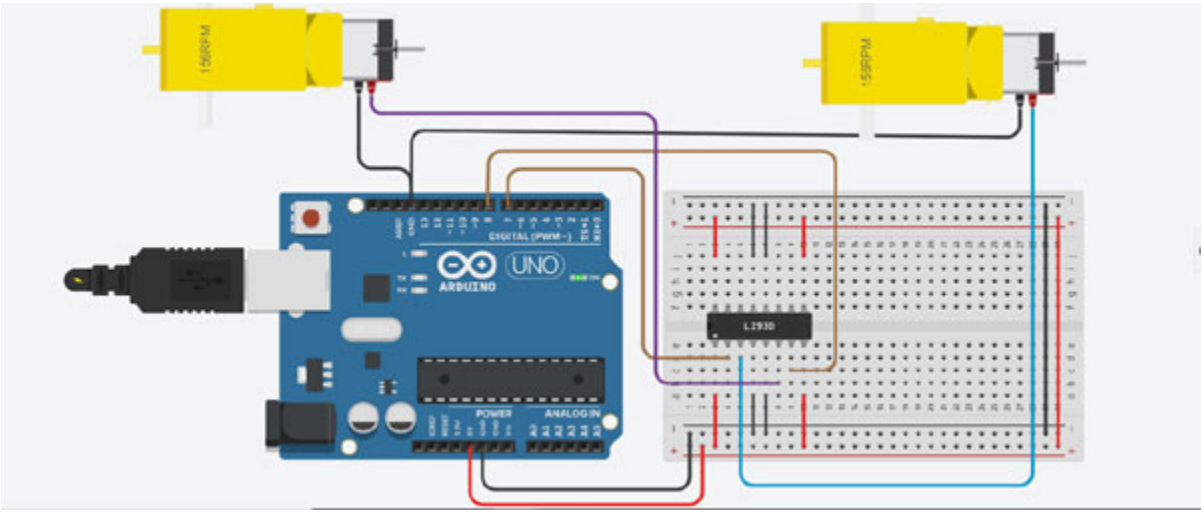


Figure 6.4: *Connections of two geared motors with L293D IC*

There are no major changes in the coding part. We can observe, in the connections, that digital pins seven and eight are associated with two different motors. The output of pin seven is given to in1 of L293D, and out1 from L293D is connected to the positive terminal of the DC motor. A similar connection is made from pin 8 to L293D and then to the second DC motor. The sketch for the recipe is given below:

```
void setup()
{
 pinMode(7, OUTPUT);
 pinMode(8, OUTPUT);
}
void loop()
{
 digitalWrite(7, HIGH);
 digitalWrite(8, HIGH);
 delay(5000); // Wait for 5000 millisecond(s)
 digitalWrite(7, LOW);
 digitalWrite(8, LOW);
 delay(5000); // Wait for 5000 millisecond(s)
 digitalWrite(7, HIGH);
 digitalWrite(8, LOW);
 delay(5000); // Wait for 5000 millisecond(s)
 digitalWrite(7, LOW);
 digitalWrite(8, HIGH);
 delay(5000); // Wait for 5000 millisecond(s)
}
```

Recipe 4: Controlling the speed of a motor using PWM

In *Chapter 1, Installing Arduino IDE*, we discussed the concept of Pulse Width Modulation (PWM). We discussed that six digital pins could be used to generate a PWM output. A PWM signal provides a value between zero to 255 on these pins. Zero corresponds to the minimum, and 255 corresponds

to the maximum output. We can use these pins to control the speed of a DC motor using Arduino.

In this recipe, we will use one of the PWM pins to control the speed of the DC motors. The circuit for the recipe is shown in *Figure 6.5*:

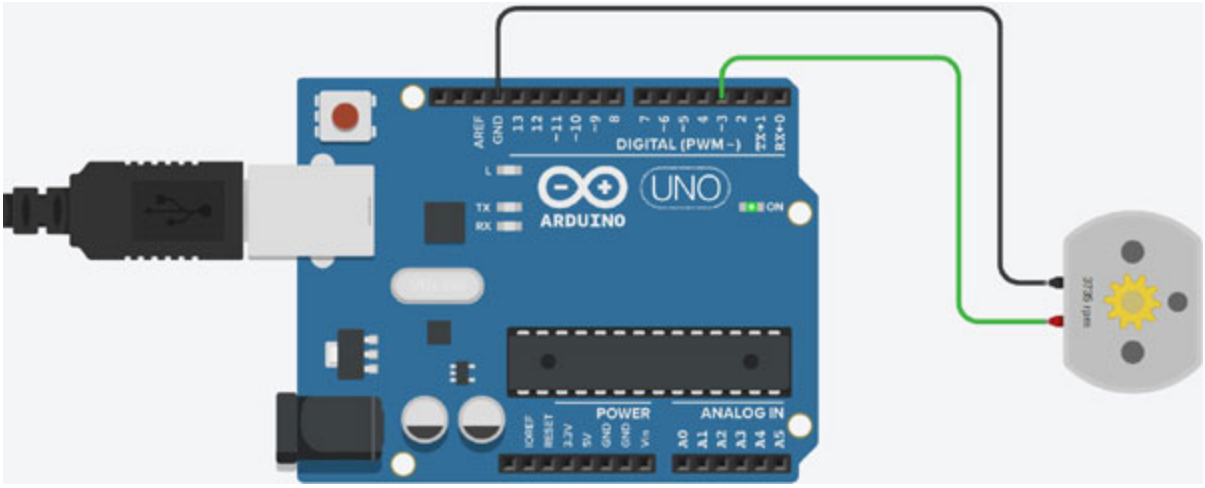


Figure 6.5: *Connecting a DC motor at the PWM pin to control the rotation speed*

The connections are similar to a DC motor in the previous recipe. The major difference is that we cannot connect the motor to any of the digital pins now, but we are restricted to only six PWM pins. The speed of the motor is directly proportional to the PWM output that we provide at the selected pin. If we provide 0, then the motor will not run, and if we provide 255, then the motor will run at the maximum speed. The speed of the motor can be linearly adjusted by selecting any number between 0 and 255.

The sketch for the recipe is as follows. In this case, we are slowly increasing the output PWM value from zero to 255. And whenever the value exceeds 250, we reset it to 0. That means the motor will slowly start increasing the rotation speed and then return to the halt state.

```
void setup()
{
 pinMode(3, OUTPUT);
}
int i = 0;
void loop()
{
```

```
  for( i = 0; i< 256; i++)
  {
   analogWrite(3, i);
   i = i+10;
   delay(100);
  }
  if (i > 250)
  {
   i = 0;
  }
 }
```

Recipe 5: Driving a 12 Volts DC motor using Arduino

In the previous recipes, we have learned the methods for controlling different DC motors that operate at 5 volts. All these motors can be supplied with the required power from the Arduino board. There are some motors that run at higher voltages. For example, the motors are designed to run at 48 or 72 Volts in electrical two-wheelers. We use a 12V DC motor for various high-power robotics applications. The Arduino board cannot run at 12 Volts, so it cannot supply 12V to such motors. In such cases, we must use the relays to connect an additional power supply that will run the desired motor.

In this recipe, we are going to discuss the concept of connecting a relay for controlling any high-powered appliances using a nerd no board. This appliance can be our AC appliances like a bulb, a fan, a tube light or a computer. This can also be a DC-operated device, like a high-powered motor. Relays are electrical devices that have an electromagnetic coil built inside. These devices have two terminals: input and output.

A relay and its internal connections are shown in *Figure 6.6*:

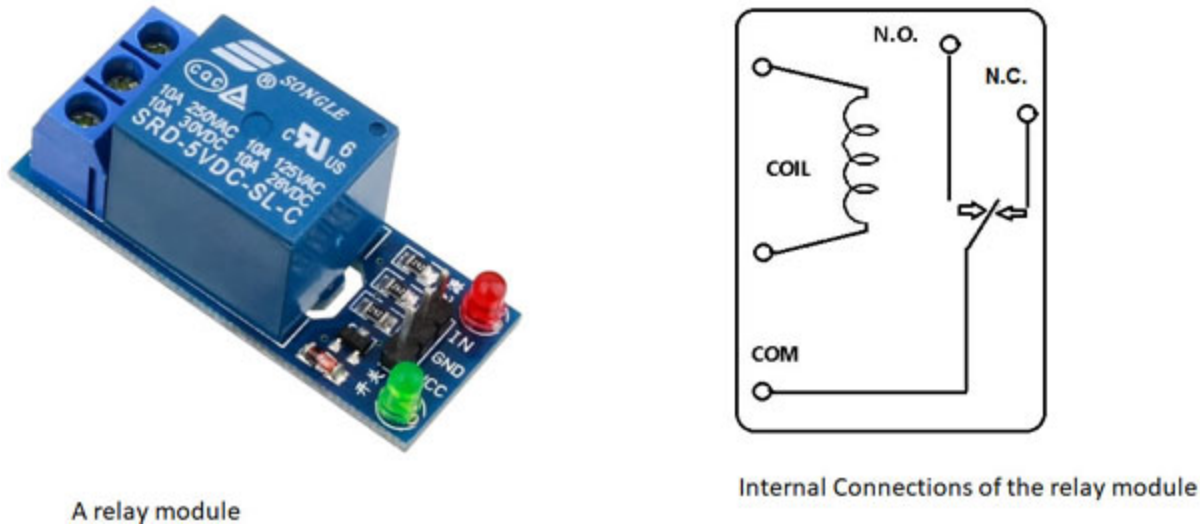


Figure 6.6: *A relay module and its internal connections*

We can observe that we have two terminals on the input side, and we can have multiple terminals, depending on the configuration, on the output side. In this example, we have 3 input pins and 3 output pins. The input pins are Ground, VCC, and Input signal. The output terminal has Normally open (NO) and **Normally Closed** (**NC**) pins. The circuit will be completed by connecting common with either NO or NC pins. If you want to make a circuit that is normally open and, when triggered, should close, then we should connect our output pins between common and NC. In this recipe, we are going to use the same combination.

The circuit connections for the recipe, including the Arduino board, 12 Volt power supply, and the relay connected to the motor, are shown in *Figure 6.7*:

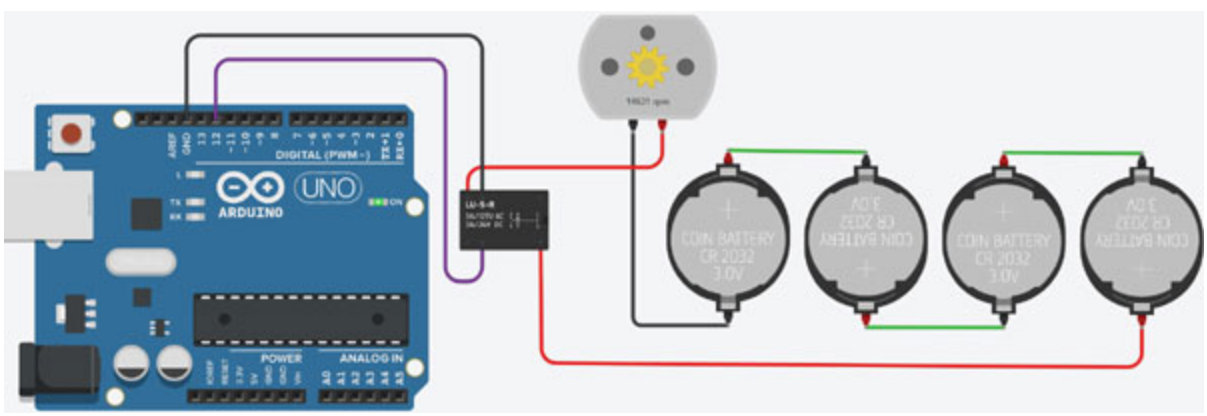


Figure 6.7: *Connections of a 12V battery with a relay to control a DC motor*

We can observe that the output is given from one of the digital pins of the Arduino board, and this output triggers the relay. The output terminal of the relay is connected to a 12 Volt battery, which is made by connecting four different 3 Volt batteries. This voltage will control the motor when the Arduino input triggers the relay.

The code for the recipe is quite simple. We do not have to mention anything specific for the relay part. We must only provide a logic high or low at one of the digital pins to trigger the relay. Once the relay is triggered, it will control the connected device with the external power supply.

```
void setup()
{
  pinMode(12, OUTPUT);
  Serial.begin(9600);
}
void loop()
{
  digitalWrite(12, HIGH);
  Serial.println("Running the motor");
  delay(5000);
  digitalWrite(12, LOW);
  Serial.println("Stopped the motor");
  delay(2000);
}
```

Recipe 6: Designing a 2-wheel drive (2WD) robot

Designing robots is one of the most used and interesting applications using an Arduino board. End Arduino enthusiasts always try to explore the field of robotics and make these moving robots. A robot requires precisely connected mechanical components controlled by the Arduino board. This recipe will discuss the requirements for designing a simple 2-wheel drive robot.

A 2-wheel drive robot has two wheels that are connected to the DC motors. Both motors are independent and can be controlled from the Arduino board. To balance the robot's body, we also need to install a few dummy wheels.

Depending on the chassis being used, we may have to use one or two dummy wheels in our robot.

Figure 6.8 shows two such two-wheeled robots that robotic enthusiasts commonly use:

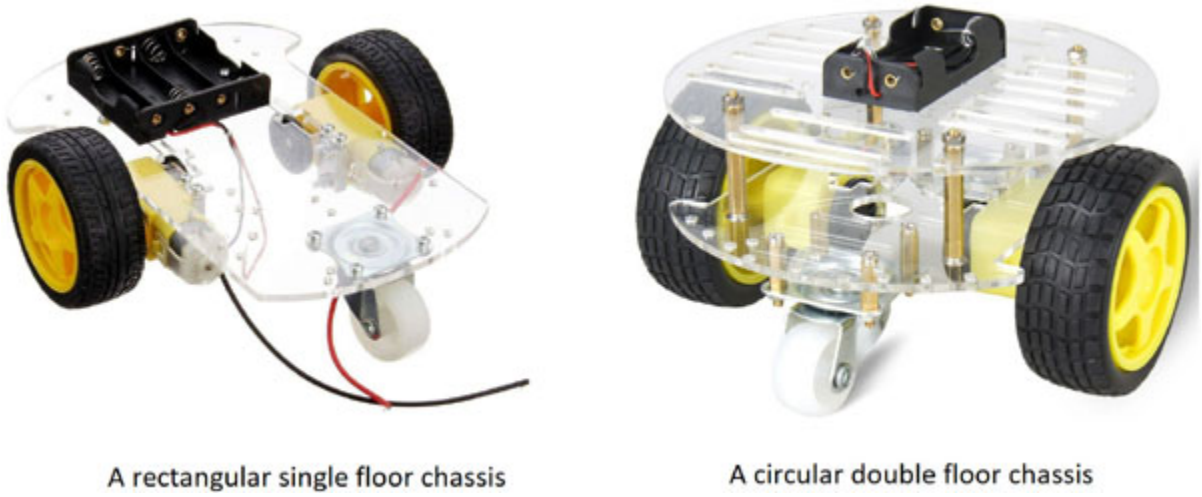


Figure 6.8: *Different types of 2-wheel robots used for hobby projects*

The circuit connections for a 2-wheel robot are very similar to recipe 3 of this chapter. We need to control two motors with an Arduino board. We can command the motors to move forward, backward, take a left turn, or take a right turn. When we make such robots, attention should be paid to robust connections because the robot will be moving and many connections can get loose. The mechanical parts should also be attached with proper care using precise tools.

We may have to use two different batteries: one for the Arduino and another for driving the motors. When we use an extra battery for driving the motors through the L293D module, we should take care that the ground of both the batteries is connected to a common terminal. The battery voltage for the motor battery should be compatible with the operating voltage of the motors. The battery used for Arduino can be in the range of 5 to 7.5 volts.

Recipe 7: Designing an obstacle avoider robot with Arduino

We designed a simple 2-wheel robot in the previous recipe. We hope that you had fun running this robot on the ground. Now, we will make some changes to the existing robot and make it autonomous. When the robot is moving on the ground, it faces various obstacles on the path. Some of these obstacles can be hazardous to the robot.

In this recipe, we will design an obstacle avoidance robot controlled by an Arduino board. The hardware design is very similar to a simple 2-wheel robot. In addition to the existing components, we will mount an ultrasonic sensor on top of the robot facing toward the front. This sensor will keep on monitoring the distance of any obstacle in front of the robot. Whenever the distance is quite small, we can change the path of the movement.

The electronic connections for the robot are shown in *Figure 6.9*. In this diagram, we have not shown the L293 motor driver. The driver should be added by the user based on the current requirements of the motors.

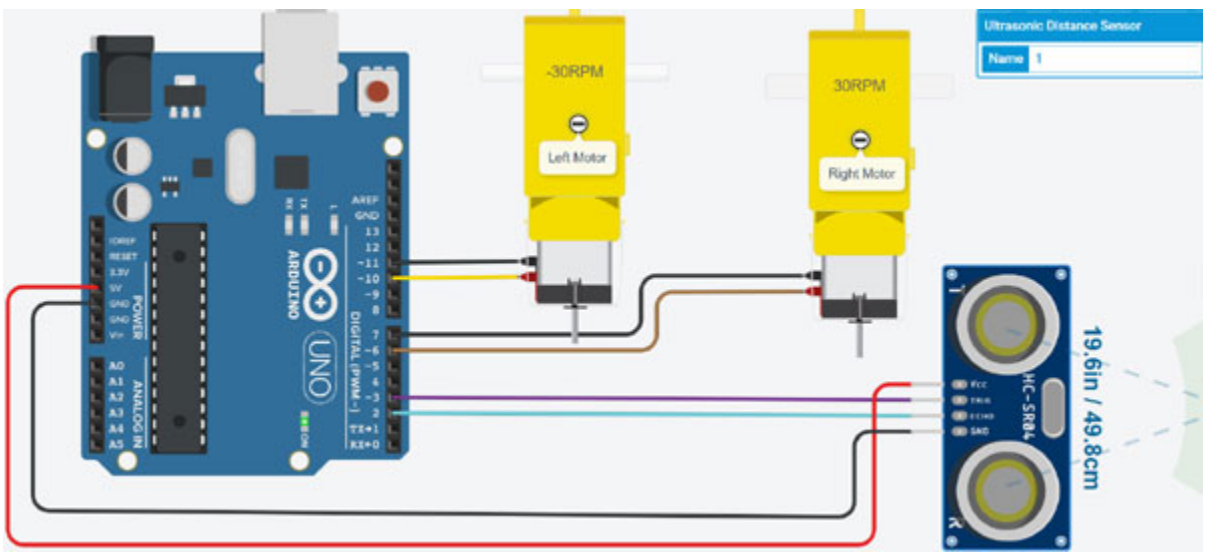


Figure 6.9: *An obstacle avoidance robot using an ultrasonic sensor*

For programming this robot, we will first measure the distance of any obstacle with the help of the ultrasonic sensor. If the object's distance is less than 50 centimeters, then we stop the robot for a while, and then we move the robot in the backward direction for 100 milliseconds. Now the robot has moved away from the detected obstacle. But if we start the robot movement again, it only goes in the direction of the obstacle. Therefore, we rotate the robot in the left or right direction by 90 degrees. Once the rotation is

completed, we move the robot forward. The code for the recipe is as follows:

```
int dist = 0;
long readUltrasonicDistance(int triggerPin, int echoPin)
{
  pinMode(triggerPin, OUTPUT); // Clear the trigger
  digitalWrite(triggerPin, LOW);
  delayMicroseconds(2);
  digitalWrite(triggerPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(triggerPin, LOW);
  pinMode(echoPin, INPUT);
  return pulseIn(echoPin, HIGH);
}
```

In the **`setup()`** function, we set the pin modes of 4 motor pins to output pins:

```
void setup()
{
  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);
  pinMode(10, OUTPUT);
  pinMode(11, OUTPUT);
}
```

The **`loop()`** function is the main function of this recipe. Initially, we calculate the distance of the obstacle and store it in the variable dist. Then we use if-else conditions to decide our operations, depending on the distance of the obstacle. If the distance is more than 50 centimeters, the robot keeps moving in the forward direction. Otherwise, the robot will follow the path discussed earlier:

```
void loop()
{
  dist = 0.01723 * readUltrasonicDistance(2, 3);
  if (dist > 50)
  {
```

```
    digitalWrite(6, HIGH);
    digitalWrite(7, LOW);
    digitalWrite(10, HIGH);
    digitalWrite(11, LOW);
    delay(100);
  }
  else
  {
    //Turn back for 200ms
    digitalWrite(6, LOW);
    digitalWrite(7, HIGH);
    digitalWrite(10, LOW);
    digitalWrite(11, HIGH);
    delay(200);
    // Turn right by 90 degrees is 500ms
    digitalWrite(6, HIGH);
    digitalWrite(7, LOW);
    digitalWrite(10, LOW);
    digitalWrite(11, HIGH);
    delay(500);
    // Move forward now
    digitalWrite(6, HIGH);
    digitalWrite(7, LOW);
    digitalWrite(10, HIGH);
    digitalWrite(11, LOW);
    delay(100);
  }
}
```

Recipe 8: Designing a line following robot to run on a black line

Now, we have a robot that can change its path if it detects any obstacle on the way. How about designing another robot that follows another interesting pattern? One of the first robots designed by robotic enthusiasts is a line

following robot. The line following robot follows a dark-colored line on a light-colored floor. Typically, the color of the floor is kept white, and the line is drawn in black color.

In this recipe, we are going to design a line-following robot with the help of li**ght-dependent resistors** (**LDR**). We had discussed the LDR sensor in recipe 1 of *Chapter 5, Sensor Integration with Arduino*.

This sensor can change the resistance based on the intensity of light falling on the sensor's surface. As we know that light colors reflect more light and dark colors reflect less light, we will utilize the same property for designing the line following robot. As we mentioned, we will use a light-colored floor and a dark-colored line. Therefore, the intensity of light coming from the floor and the line will differ. These intensities will generate different values of resistance of LDR. Sometimes the natural light is enough to get reflected from the surface. In contrast, in some cases, we may attach additional leads in the downward direction to provide more light to be reflected from the surface.

The circuit connections for the recipe are shown in *Figure 6.10*:

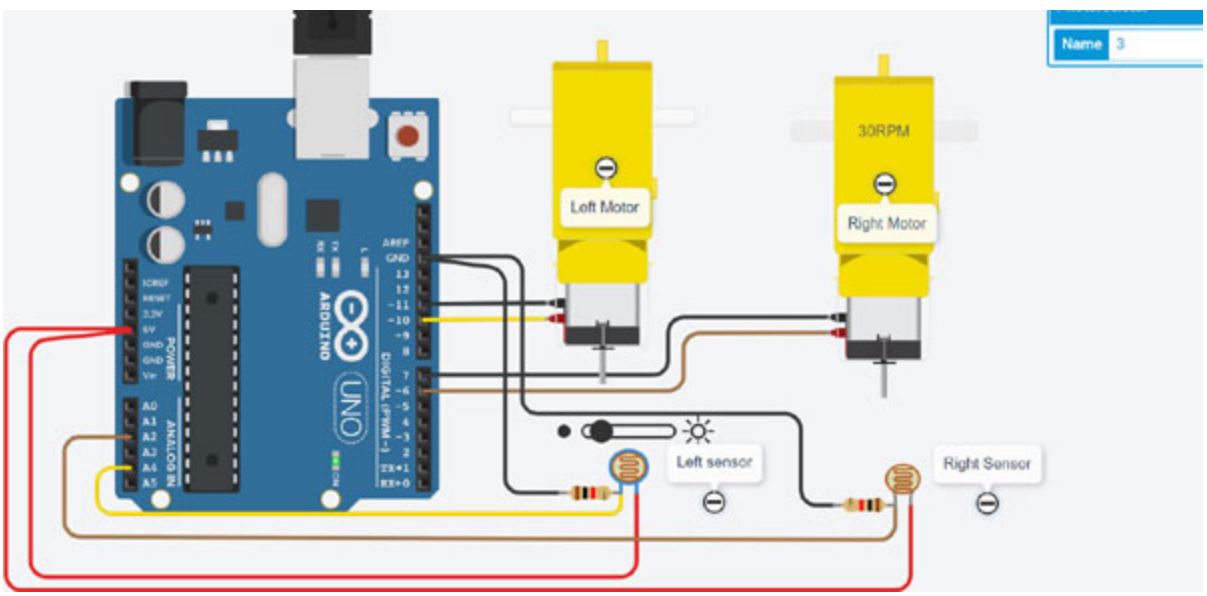


Figure 6.10: *Connections for a line-following robot*

In this figure, we have used two LDRs. One of the sensors is mounted on the left side of the robot facing downward, and the other is on the right side. Both these sensors detect the light intensity coming from the floor or line.

The outputs of these LDRs are connected to two separate analog input pins. We must check the intensity using the serial monitor coming out from the light surface and the dark line. On our observations, we can decide a central value to be our threshold value. For example, if the value received on the floor is 500 and the value received on the line is 700, we can decide that the threshold value will be somewhere around 600. These observations will completely depend on the environment in which you work, and you must set this threshold by observing the values of light intensities on two different surfaces.

The coding for the robot is quite easy. Let us understand the complete sketch in parts. In the **`setup()`** function, we set the pin modes for 4 motor pins as **`OUTPUT`** and two analog pins as **`INPUT`**:

```
void setup()
{
  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);
  pinMode(10, OUTPUT);
  pinMode(11, OUTPUT);
  pinMode(A4, INPUT);
  pinMode(A2, INPUT);
}
```

We declare 2 integers, L, and R, to store the analog inputs coming from that respective LDRs:

```
int L, R;
```

The loop function is the main function for this recipe. We simultaneously read the value of L and R. If both the sensor values are more than the threshold, then the robot keeps moving in the forward direction. If any one of the sensors detects the dark color line, that particular motor is stopped. For example, if the dark-colored line is detected on the left sensor, then we will stop the left motor of the robot. As soon as the left motor stops, the motion of the left part of the robot will cease. But the right motor is still moving, and therefore, effectively, the robot will start moving towards the left. And after a certain moment, the robot will realign itself on the line, and

now the left sensor will also be on the floor and note on the line. Once the robot is realigned on the floor, it will start moving forward.

```
void loop()
{
  L = analogRead(A4);
  R = analogRead(A2);
  // Both sensors on the white surface
  if (L > 500 && R > 500)
  {
    digitalWrite(6, HIGH);
    digitalWrite(7, LOW);
    digitalWrite(10, HIGH);
    digitalWrite(11, LOW);
    delay(100);
  }
  // Left sensor on the black line
  else if (L < 500 && R > 500)
  {
    digitalWrite(6, HIGH); // Right will move
    digitalWrite(7, LOW);
    digitalWrite(10, LOW); // Left will stop
    digitalWrite(11, LOW);
    delay(100);
  }
  // Right wheel on the black line
  else if (L > 500 && R < 500)
  {
    digitalWrite(6, LOW); // Right will stop
    digitalWrite(7, LOW);
    digitalWrite(10, HIGH); // Left will move
    digitalWrite(11, LOW);
    delay(100);
  }
  else
```

```
  {
   //Both motors move forward
   digitalWrite(6, HIGH);
   digitalWrite(7, LOW);
   digitalWrite(10, HIGH);
   digitalWrite(11, LOW);
   delay(200);
   }
 }
```

Recipe 9: Controlling the angle of movement of a servo motor

We have discussed DC motors and designed a few applications using the same. Now, let us discuss a new type of motor, and that is a servo motor. A servo motor is also controlled with DC but has a completely different way of operation. A server motor is not designed to rotate continuously but to rotate from one point to another with a precise recorded movement. Typically, a servo motor is designed to rotate at an angle of 180 degrees. Therefore, the rotation of a servo motor is restricted to a semicircle only.

In this recipe, we will learn about the connections and the coding involved to control the angle of movement of a servo motor using the Arduino board. The connections for this recipe are shown in *Figure 6.11*. The servo motor has three pins. One pin is dedicated to VCC, and one is for the ground. The third pin is the signal pin, which is the most important pin for motor movement. The signal pin can be connected to one of the digital pins of the Arduino board, and we provide the rotation angle through this pin.

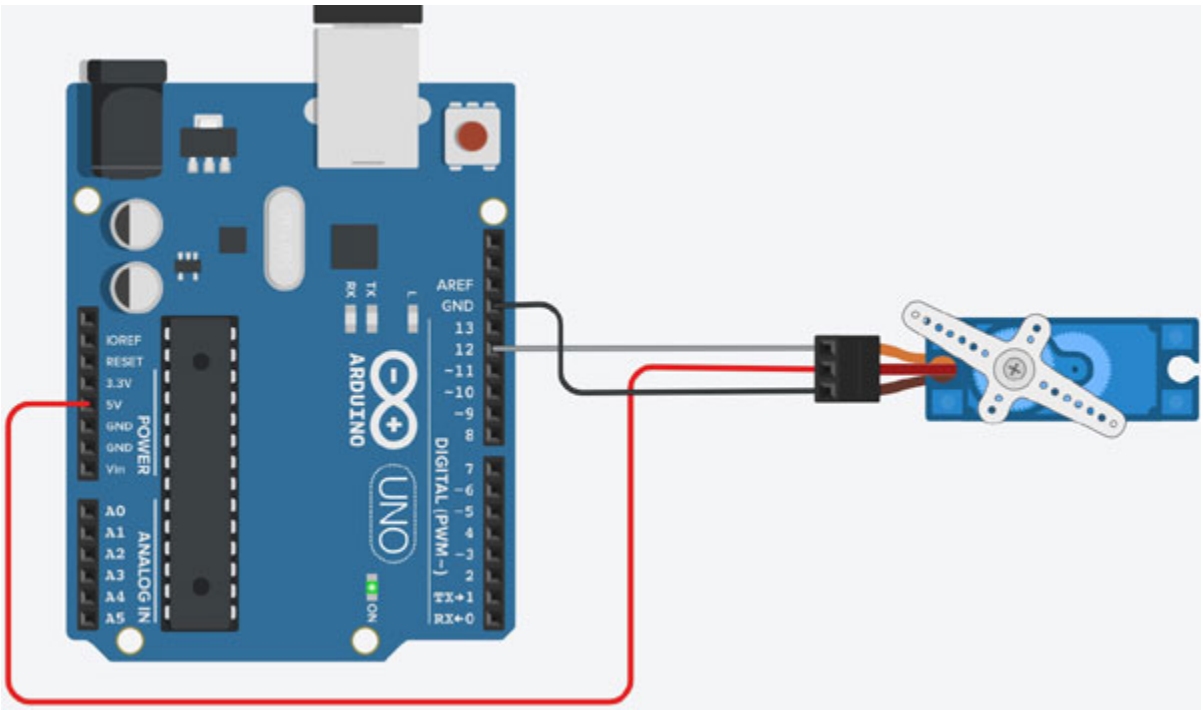


Figure 6.11: *Connections of a servo motor with an Arduino board*

We need a dedicated library to define various functions related to servomotor operation. We use the **`servo.h`** library file in the coding. There are various functions inbuilt inside this library, and we will use some of them based on the requirement. The first function that is used to mention the pin on which the servo motor is attached to the Arduino board is **`server.attach`**. In our sketch, we used PIN 12 for attaching the servo motor. The **`servo.write`** function is used to rotate the servo motor to a particular angle. In this sketch, we are slowly increasing the angle from zero to 180 degrees and then bringing it back to zero. We have used 2 for loops for increment and the decrement of the rotation angle.

```
#include <Servo.h>
int Angle = 0;
Servo servo_12;
void setup()
{
 Serial.begin(9600);
 servo_12.attach(12, 500, 2500);
}
```

```
void loop()
{
 // Increase the angle slowly
 for(Angle = 0; Angle < 180; Angle++)
 {
  Serial.println(Angle);
  servo_12.write(Angle);
  delay(10);
 }
 // Reduce the angle slowly back to 0
 for(Angle = 180; Angle > 0; Angle--)
 {
  Serial.println(Angle);
  servo_12.write(Angle);
  delay(10);
 }
}
```

Recipe 10: Controlling the movements of a servo with a potentiometer

In the previous recipe, we learned how to control the angle of rotation for a servo motor. Now, let us control the rotation of a servo motor with our hands. All of us might have seen it. The rotatory, which is a switch is used to control the speed of a fan or the temperature of a microwave and similar other appliances. All these rotary switches are made up of potentiometers. A potential metre is made by using two variable resistors. A potentiometer has three terminals. The central terminal is the output, and the two corner terminals are connected to the two voltage signals.

To discuss the rotation of a servo motor by rotating a potential metre. The circuit for the recipe is shown in *Figure 6.12*. We have connected one end of the potentiometer to the ground and the other end to 5V. The input available at the analog pin ranges from 0 to 1023. This input will control the angle of rotation for the servo motor.

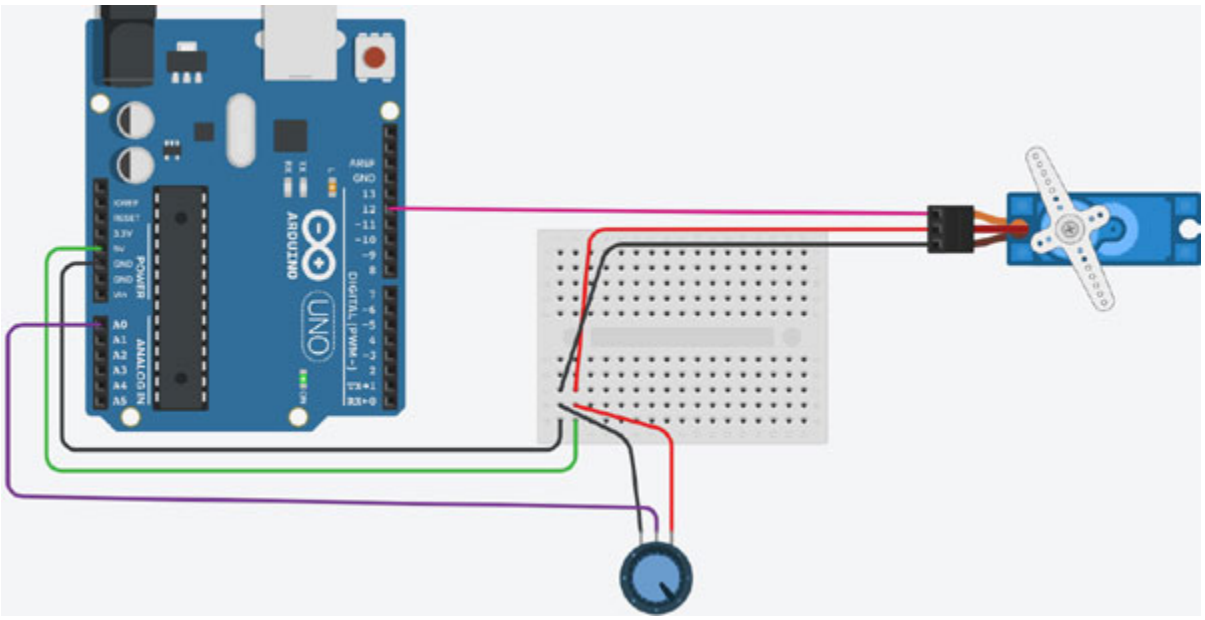


Figure 6.12: *A potentiometer controlling the angle of rotation of a servo motor*

Now, let us discuss the code. In the sketch, we read the analog input from analog pin A0 and store it in a variable. Then we use the map function to map the input range of the variable angle, from 0 to 1023 to an output range of 0 to 180. Therefore, 0 will be mapped to 0, and 1023 will be mapped to 180. All the other values are mapped linearly. The sketch for the recipe is as follows:

```
#include <Servo.h>
int Angle = 0;
Servo servo_12;
void setup()
{
 pinMode(A0, INPUT);
 Serial.begin(9600);
 servo_12.attach(12, 500, 2500);
 pinMode(LED_BUILTIN, OUTPUT);
}
void loop()
{
 Angle = analogRead(A0);
 Serial.println(Angle);
```

```
  servo_12.write(map(Angle, 0, 1023, 0, 180));
  digitalWrite(LED_BUILTIN, HIGH);
  delay(10); // Delay a little bit to improve simulation
  performance
}
```

Recipe 11: A smart dustbin that opens the lid on detecting a person around it

As the COVID-19 pandemic struck the globe, social distancing and safety became the prime concerns for everyone. People became more cautious about touching different surfaces, especially in public places. Covered dustbins are placed in almost every public place. To avoid touching them, we must design a smart dustbin in this recipe.

The smart dustbin designed in this recipe will detect any person near the opening and open the lid itself to let them use the dustbin. This is quite a useful application in real-world problem-solving. We use an ultrasonic sensor to detect any object and mount it on the front side of the dustbin.

Figure 6.13: *A smart dustbin designed using an ultrasonic sensor and servo motor*

Figure 6.13 shows the arrangement made for the purpose. We can observe that an ultrasonic sensor is mounted on the front side, and the lid is closed.

To open the lid, we have attached a servo motor to the top of the dustbin. The motor is attached with a strong string, pulling the lid up to provide access to the dustbin.

Figure 6.14 shows the circuit connections for the smart dustbin. We have attached the echo and trigger pins of the ultrasonic sensor to digitals PINs 2 and 3. The servo motor is controlled by the output pin 12.

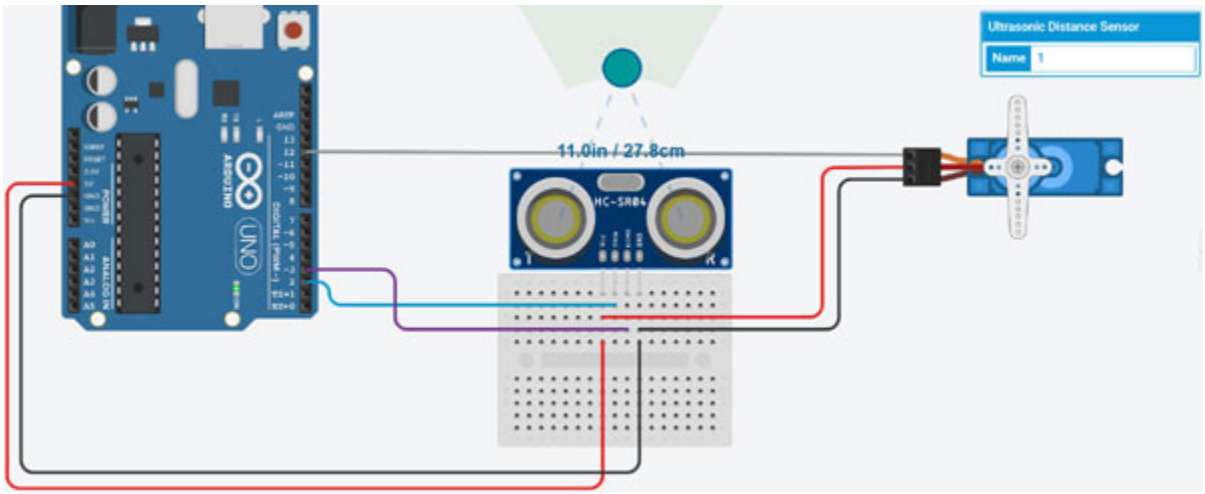


Figure 6.14: *Circuit connections for a smart dustbin*

The coding for the recipe is simple, but we require some custom setup based on the implementation area. We need to fix the distance of the object to be detected. In this sketch, we consider the distance to be 30cm, and the user can change it per the requirements. Similarly, we are rotating the servo motor by 90 degrees and assuming that the lid will open with this rotation; the user can change the angle based on their setup.

```
#include <Servo.h>
int Angle = 0;
Servo servo_12;
int dist = 0;
long readUltrasonicDistance(int triggerPin, int echoPin)
{
  pinMode(triggerPin, OUTPUT); // Clear the trigger
  digitalWrite(triggerPin, LOW);
  delayMicroseconds(2);
  digitalWrite(triggerPin, HIGH);
```

```
  delayMicroseconds(10);
  digitalWrite(triggerPin, LOW);
  pinMode(echoPin, INPUT);
  return pulseIn(echoPin, HIGH);
}
void setup()
{
  pinMode(A0, INPUT);
  Serial.begin(9600);
  servo_12.attach(12, 500, 2500);
  pinMode(LED_BUILTIN, OUTPUT);
}
void loop()
{
  dist = 0.01723 * readUltrasonicDistance(2, 3);
  if(dist < 30)
  {
    servo_12.write(90);
    delay(5000);
  }
  else
  {
    servo_12.write(0);
  }
}
```

Recipe 12: Automated parking boom barrier

Traffic management is one of the crucial issues in urban areas. It is a must for managing the flow of traffic and for security. Most of us would have seen automatic boom marriers installed in parking spaces, toll barriers, and entrances and exits of some complexes. These barriers keep the path shut for traffic, and they open up to give access to the traffic. Refer to the following figure:

Figure 6.15: *An automated boom barrier*

Sometimes, these barriers are operated manually, and sometimes they are automated. *Figure 6.15* shows one such barrier installed at the entry of a particular society. The working principle behind the design of such barriers is very similar to the last recipe. We install ultrasonic sensors facing the front to detect any incoming vehicles. Once a vehicle is detected within the specific range, a high-powered servo motor is triggered to raise the barrier. The user can specify the angle of rotation for the motor. In place of servo motors, DC motors are better options for such high-powered applications. We calculate the time required to raise the barrier up to a certain height and then run the motor for that much time to raise the barrier.

The circuit connections and the coding for this recipe are similar to the ones for the smart dustbin recipe. You can modify the same with additional hardware rearrangements to design this recipe.

Recipe 13: Designing a robotic hand to pick a glass of water

Life is hard for people having amputated limbs. They cannot perform simple daily routine activities. In this recipe, we discuss the design of a robotic hand for a person with one amputated hand. The hand is designed using an Arduino board, flex sensors, and servo motors. The hand will mimic the action of the other hand and help the person perform some actions that require two hands.

In this recipe, we will design the electronic controller for a robotics arm. This controller is controlled by the actions of one hand and mimics them on

the other hand.

Figure 6.16 shows the construction of such a robotic hand. The complete robotics hand will have 5 flex sensors on the input side and 5 servo motors controlled by these sensors. We can observe 5 motors controlling the movements of 5 fingers.

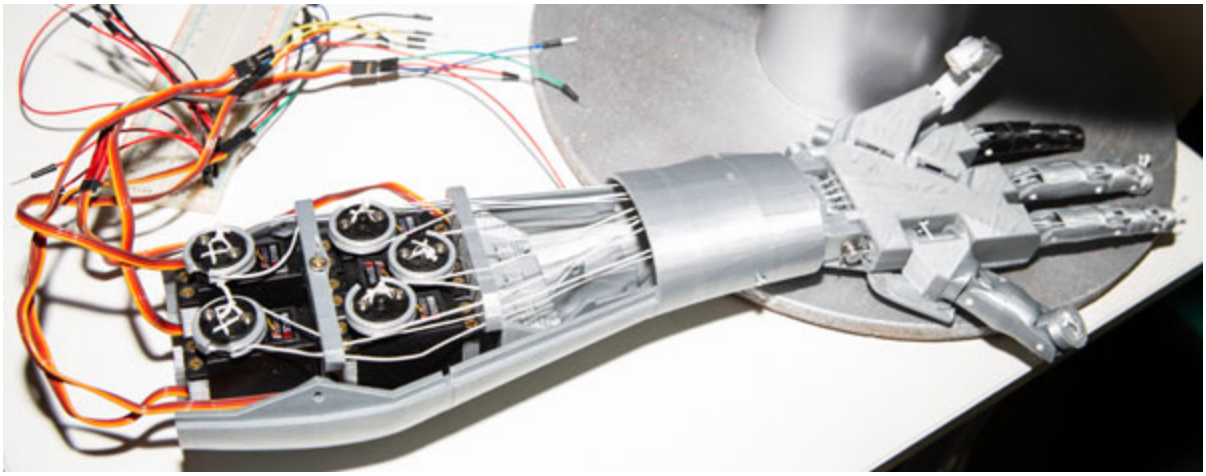


Figure 6.16: *A robotic hand construction using servo motors and movable fingers*

A flex sensor is in the form of a strip. It is made up of the variable resistive surface. The resistance will change with the bend in the strip. Therefore, the sensor's resistance is linearly proportional to the degree of bend in the strip. We make a glove and attach five sensors to each of the fingers. Now, when we perform any action through the hand with a glove, all these sensors are bent to different degrees. Therefore, we get unique resistance values for each of these sensors.

For simplicity, let us consider one flex sensor and servo motor combination. We must replicate this combination five times to configure the whole system. *Figure 6.17* shows the circuit connections for the combination. The input from the flex sensor is given at one of the analog pins. The angle of rotation for the servo motor will be controlled by the input values of the flex sensor.

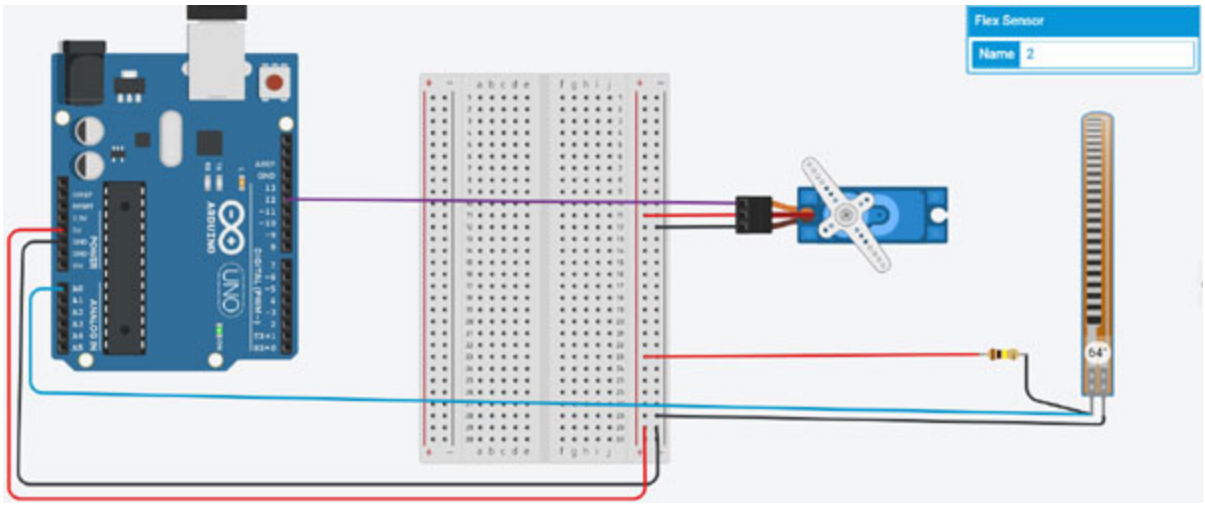


Figure 6.17: *A part of a robotic hand designed using a flex sensor and servo motor*

We must observe the analog pin values before using the **`map()`** function for coding. The output of the flex sensor doesn't produce the full range of 0 to 1023 at the analog pin. So, we must manually bend the sensor and observe the minimum and maximum values received at the analog pins. In this example, the values are 236 and 634. Therefore, we'll use the following statement to map the flex sensor values to the servo motor angles:

```
servo_12.write(map(Angle, 236, 634, 0, 180));
```

The complete code for the recipe is as follows:

```
#include <Servo.h>
int Angle = 0;
Servo servo_12;
void setup()
{
  pinMode(A0, INPUT);
  Serial.begin(9600);
  servo_12.attach(12, 500, 2500);
  pinMode(LED_BUILTIN, OUTPUT);
}
void loop()
{
  Angle = analogRead(A0);
  Serial.println(Angle);
```

```
  servo_12.write(map(Angle, 236, 634, 0, 180));
  digitalWrite(LED_BUILTIN, HIGH);
  delay(10); // Delay a little bit to improve simulation
  performance
}
```

Recipe 14: Controlling a stepper motor with Arduino

We have discussed a DC motor and a servo motor in our earlier recipes. Now we are going to talk about a new type of motor: a stepper motor. A stepper motor is also operated with a direct current. The working principle of a stepper motor is different from a DC or servomotor. It is a brushless DC motor that divides the complete rotation into a number of equal steps. The motor's position can be controlled to move and hold at one of these steps. We do not require any position sensor for feedback as long as the motor is correctly sized to the application with respect to torque and speed.

Figure 6.18 shows a stepper motor commonly used with an Arduino board and its connection with the board. We can observe that we need to have a motor driver between the motor and the Arduino board to control the motor correctly.

The motor used in this recipe is 28BYJ-48. It works at 5V and, therefore, is a perfect motor for Arduino-based projects. The driver IC is ULN2003. The complete system works with a combination of motor and driver. **Stepper motors are widely used in printers.** Refer to the following figure:

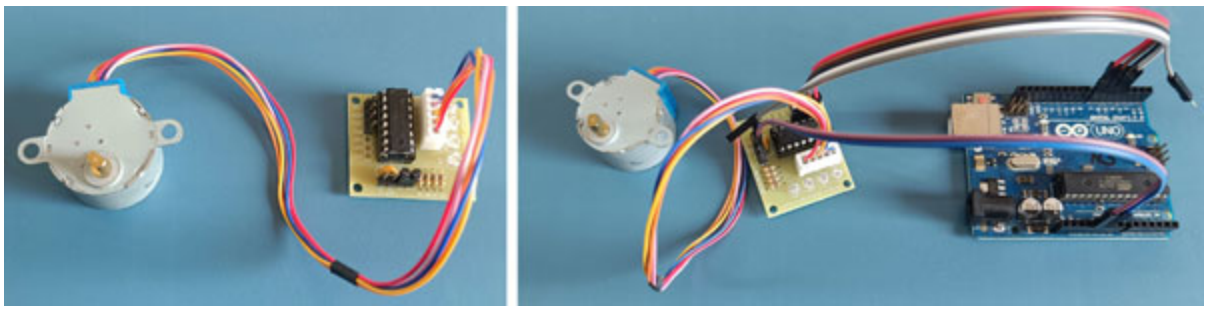


Figure 6.18: *A stepper motor and its connections with an Arduino board*

The stepper motor used in this recipe has four equal steps for one full rotation. We need to provide input in a sequence to all these steps to make the rotation. Typically, we provide the input in an incremental or

decremental manner from the first step to the last. For example, sequence 1000 implies that we are giving high input to step one and low input to the remaining ones. If we want to rotate the motor in a clockwise direction, our next input case should be 1100. With this input, we are preparing the second step for the rotation. In the next step, we can provide 0100 to ensure that step one has stopped and the next one is still activated. Going in the same manner, our next sequence of inputs will be 0110, 0010, 0011, and 0001. In between every input case, we should also provide a delay.

The sketch for the recipe is as follows. Initially, we set up four digital pins as the output pins. These pins are connected to the motor driver. In the **`loop()`** function, we declare our input sequences to run the motor in the clockwise or anticlockwise direction.

```
int mx_steps = 512;
void setup()
{
 pinMode(6,OUTPUT);
 pinMode(7,OUTPUT);
 pinMode(8,OUTPUT);
 pinMode(9,OUTPUT);
}
void write(int W,int X,int Y,int Z)
{
 digitalWrite(6,W);
 digitalWrite(7,X);
 digitalWrite(8,Y);
 digitalWrite(9,Z);
}
void onestep()
{
 write(1,0,0,0);
 delay(50);
 write(1,1,0,0);
 delay(50);
 write(0,1,0,0);
```

```
  delay(50);
  write(0,1,1,0);
  delay(50);
  write(0,0,1,0);
  delay(50);
  write(0,0,1,1);
  delay(50);
  write(0,0,0,1);
  delay(50);
  write(1,0,0,1);
  delay(50);
}
void loop(){
int i = 0;
while(i<max_steps)
{
  onestep();
  i = i+1;
  }
}
```

Recipe 15: Controlling a DC motor with an encoder with Arduino

DC motors are quite popular in the field of robotics and automation. The **Rotations Per Minute** (**RPM**) of these motors are quite high. Therefore, we attach gears in front of the motors for slow and powerful movement. *Figure 6.19* shows one such DC motor with ger built on the front side. This motor has not only a gearbox but also an encoder fixed inside the motor.

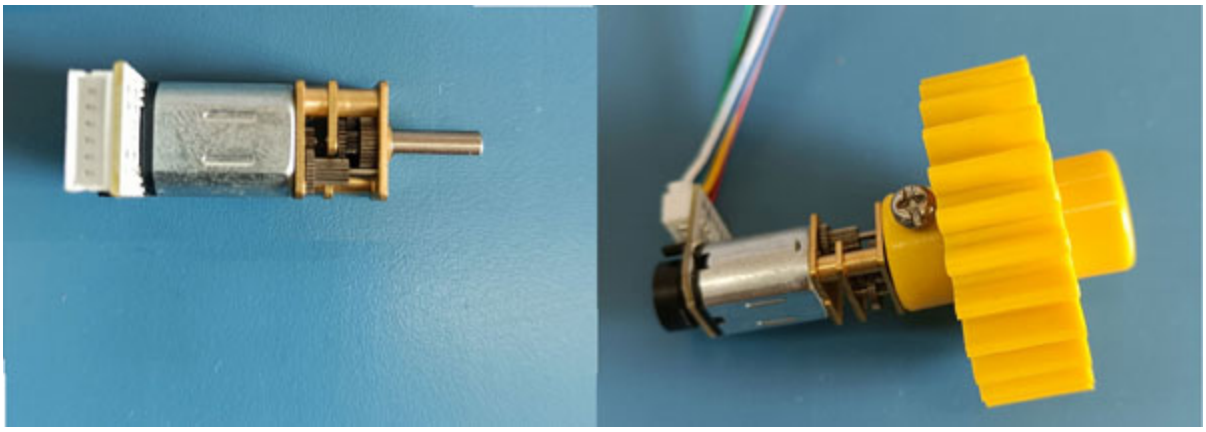


Figure 6.19: *A geared DC motor with a built-in encoder connected with a wheel to observe rotations*

An encoder is an electromechanical device that provides an electrical signal used for speed and/or position control. In other words, we can say that we get encoded numbers for each movement of the motor. The count of these numbers increases for every clockwise rotation and decreases for every anticlockwise rotation. In the same figure, we have attached a wheel on top of the motor to observe the clockwise/anticlockwise rotation.

The encoder is attached to the backside of the motor. It is soldered firmly and provides the connections to read the encoder values from the motor. There are a total of 6 pins on the connection terminal.

Let us understand the purpose of these pins through *Table 6.1*:

Pin code	Full name	Purpose
M1	Motor Terminal 1	The negative terminal of the motor, is connected to the H bridge
M2	Motor Terminal 2	The positive terminal of the motor is connected to the H bridge
VCC	Power supply	Power supply from the controller (Arduino)
GND	Ground pin	Ground from the controller
C1	Encoder Output A	To be connected to Arduino
C2	Encoder Output B	To be connected to Arduino

Table 6.1: *Different pins and their purposes in a DC motor with encoders*

Since this is a geared motor, it will require H-bridge (L293D) to run. The circuit connections for the recipe are shown in *Figure 6.20*:

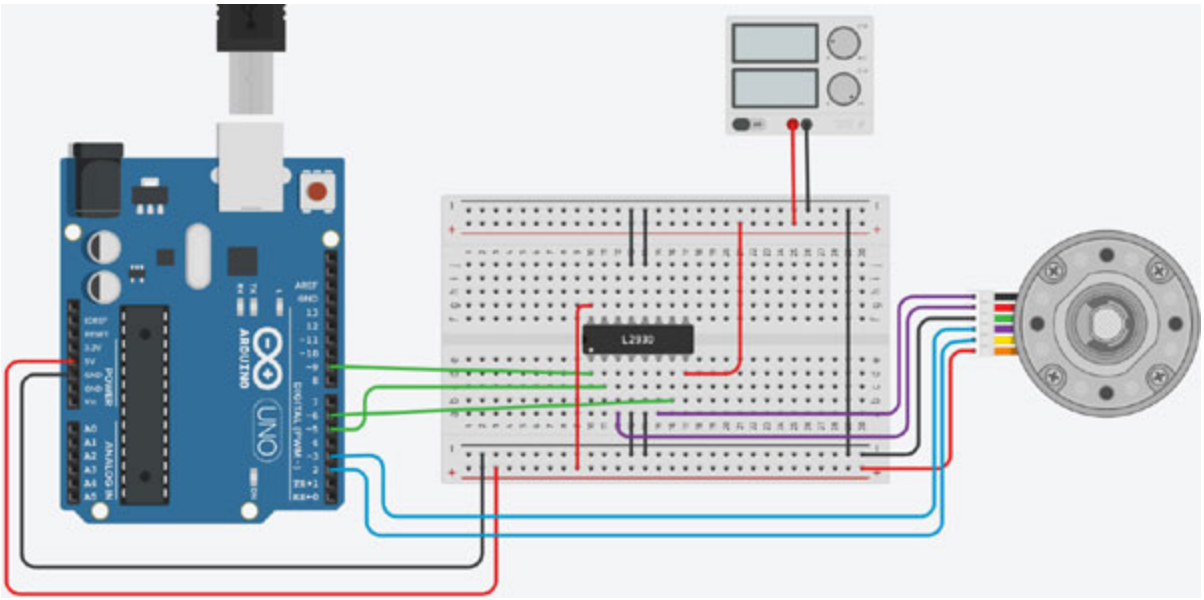


Figure 6.20: *Circuit connections for a geared DC motor with encoders*

In this recipe, we will count the encoder values coming from the motor and convert them into the rotations of the gearbox output. We must read the datasheet of the motor and find the gear ratio and the number of encoders generated per revolution. In this recipe, we are using "*N20 6V 100RPM Micro Metal Gear Motor with Encoder*". This motor has a gear ratio of 1:150 and generates 48 codes per revolution. Therefore, every revolution from the DC motor is reduced 150 times at the gearbox's output. In coding, we use the serial monitor to print the number of revolutions. We can also observe the same by counting the revolutions of the wheel.

Let us understand the sketch for the recipe now. Initially, we define the variables associated with different pins and signals required:

```
const int motorDir = 5;
const int motorPWM = 6;
const int EnPin = 9;
const int EnPinA = 2;
const int EnPinB = 3;
volatile int EnPos = 0;
boolean EnALast = LOW;
```

Then, we calculate the wheel's revolutions ratio to the actual RPM by dividing the RPM by the gear ratio and countable events per revolution. Here we had 360 degrees in one rotation, 1:150 as the gear ratio of the N20 motor and 48 countable Events Per Revolution:

```
// encoder value change motor turn angles
const float ratio = 360/150/48.;
```

Then, we declare a function to move the motor in one direction with a given velocity:

```
void doMotor(bool dir, int vel)
{
  digitalWrite(motorDir, dir);
  analogWrite(motorPWM, vel);
}
```

In the **`setup()`** function, we define the pin modes for all the pins:

```
void setup()
{
  Serial.begin(9600);
  pinMode(motorDir, OUTPUT);
  pinMode(EnPin, OUTPUT);
  pinMode(EnPinA, INPUT);
  pinMode(EnPinB, INPUT);
  digitalWrite(EnPinA, HIGH);
  digitalWrite(EnPinB, HIGH);
}
```

The **`loop()`** function is the main function for the recipe. We rotate the motor at 100 RPM using the **`doMotor`** function. Then, we increase the encoder values if we detect clockwise rotations and decrease it for anticlockwise rotations:

```
void loop()
{
  digitalWrite(EnPin, HIGH);
  doMotor(HIGH, 100);
  boolean EnANow = digitalRead(EnPinA);
  if((EnALast ==HIGH) && (EnANow == LOW))
```

```
  {
   if(digitalRead(EnPinB) == LOW)
   {
    EnPos++;
   }
   else
   {
    EnPos--;
   }
   Serial.println(float(EnPos)*ratio);
  }
  EnALast = EnANow;
 }
```

Conclusion

In this chapter, we studied different types of motors. We discussed a simple DC motor and a lightweight servo motor. Using these motors, we designed simple robots as well. We integrated the sensors with the robots and designed some autonomous robots. The servo motor also helped us in developing some COVID-19-related projects. The combination of sensors and motors really helped us in developing full projects. At the end, we learnt the methods of controlling stepper motors and DC motors with an encoder. These motors are used for very specific applications and must be studied in depth.

In the next chapter, we will learn the protocols and connections for **Serial Peripheral Interface** (**SPI**) and I2C communication. We'll discuss various applications using these protocols and develop some projects using the same with an Arduino board.

Points to remember

- Any motor should not be operated at a voltage higher than the prescribed range.

- Servo motor connections must be checked carefully before being switched on.
- The motors should not be loaded with heavy load or force to oppose the rotations.
- Motors drivers should have a separate power supply.
- The ground of all the power supplies in the system must be connected to one common point.
- Soldering, if required, should be done carefully.

Multiple choice questions

1. **A DC motor has ____ terminals.**
    a. 1
    b. 2
    c. 3
    d. 4
2. **A servo motor can be connected to the ____ pin of the Arduino for controlled movements.**
    a. Digital
    b. VCC
    c. Interrupt
    d. Analog
3. **A servo motor can rotate up to an angle of ____ degrees.**
    a. 100
    b. 90
    c. 180
    d. 360
4. **How many pins are present in the Motor Driver (L293D) Module?**
    a. 12

b. 16
c. 14
d. 18

5. **How many different motors can be controlled in one direction by a single L293D?**

a. 1
b. 2
c. 3
d. 4

Answers

1. **b**
2. **a**
3. **c**
4. **b**
5. **d**

CHAPTER 7

I2C and SPI Communication

Introduction

This section and chapter will introduce you to different communication options with other boards and devices. I2C is also known as IIC, a short form for Inter-Integrated Circuit. We can connect one Arduino board with another or with any other device that supports I2C communication. This communication uses fewer numbers of wires. In addition to I2C, we have the support of SPI (Serial Peripheral Interface) on the Arduino board. SPI is a serial communication protocol. In this communication, devices act like masters and slaves. That means one device can control the others.

This chapter will cover the concepts of I2C and SPI communication protocols with related recipes. We will learn how to connect two Arduino boards and communicate between these two protocols using the same.

Structure

- Connect two Arduino boards to set up I2C communication
- Connecting I2C adapter to Arduino
- Controlling a 16x2 LCD with I2C protocol
- Displaying data on an OLED screen
- Digital voltmeter design using Arduino
- Understanding SPI communication on an Arduino board
- Communicate between two Arduino boards using SPI communication

Objectives

By the end of this chapter, the learner will be able to do the following:

- Communicate between two Arduino boards using the I2C protocol

- Display long messages on LCD using the I2C protocol
- Design complex projects with fewer wires using the I2C protocol
- Learn and establish SPI communication on an Arduino board

Recipe 1: Connect two Arduino boards to set up I2C communication

Inter-Integrated Circuits (**I2C**) communication is sometimes called a **2-wire interface** (**TWI**). Therefore, we should remember that I2C and TWI are the same protocol, which was invented by Philips. As the name suggests, we require only two wires to communicate using this protocol. However, for practical implementation, we will require four wires. There should not be any confusion here since the additional two wires are used to provide ground and supply voltages. Therefore, calling it a two-wire communication is not wrong.

In this recipe, we will learn the working principle of the I2C protocol and control one Arduino board with another Arduino board. There are two pins assigned to any board supporting the I2C protocol:

- **SCL (clock line):** It is used to synchronize all data transfers over the I2C bus.
- **SDA (data line):** It is used to send the data.

On Arduino UNO, analog pins A4 and A5 are assigned for I2C communication. Therefore, on both the Arduino boards, we'll use these two pins only. On Arduino UNO, pin A4 is SDA and A5 is SCL. And we need to use the `Wire.h` library to access the I2C features and related functions. I2C communication is not restricted to two devices only. We can have more than two devices involved and control all of them with one master device. Therefore, this communication is like a master-slave protocol, where one device acts like a master and the others can be slaves to its commands. Refer to the following figure:

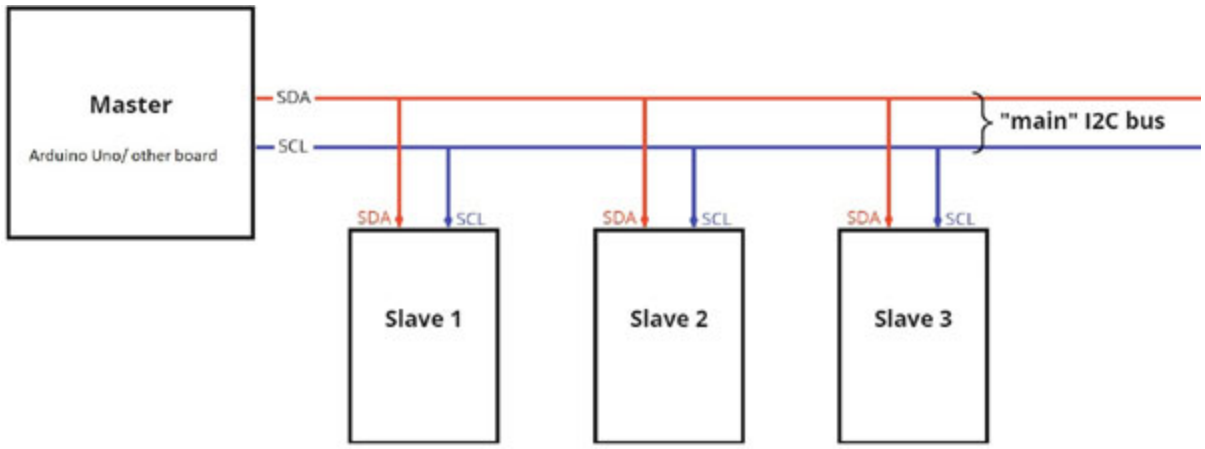


Figure 7.1: *The setup for a master-slave configuration using the I2C protocol*

Figure 7.1 shows the architecture connected multiple devices in the master-slave configuration using the I2C protocol. In this setup, we can observe that one master device can send commands to multiple slave devices.

The next important question that pops into our minds is, "*How do we send the data of thousands or millions of bits using only two wires?*" Actually, not even two; we have only one wire dedicated to the data transfer; the other is used for synchronizing the clock. Since we have only one wire for data transfer, the data will be sent one bit at a time. This type of data transfer is called serial transmission. The SDA and SCL wires work together in the whole communication process. *Figure 7.2* shows the various stages in the process:

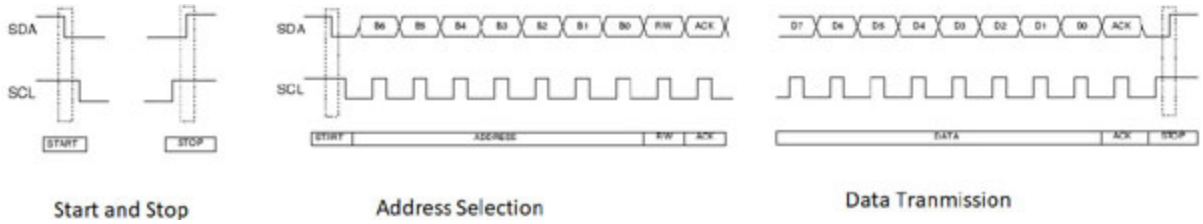


Figure 7.2: *Various steps involved in the I2C process*

Initially, we should understand that the complete process is started with the START signal and ends with the STOP signal. In between the start and stop, we have two major operations: address selection and data transmission. One byte (8-bit) of data is sent at a time, and then the acknowledgment is sent for the same. These data packets are sent in every clock cycle. Therefore, everything is synchronized with the clock pin.

The simple objective of this recipe is to send a signal from the master device and to give the command to the slave device. We will have a pushbutton connected to the master device and an LED to the slave device. Whenever we press the pushbutton on the master device, the LED will turn on at the slave device. Let us first look at the circuit diagram for this in *Figure 7.3*:

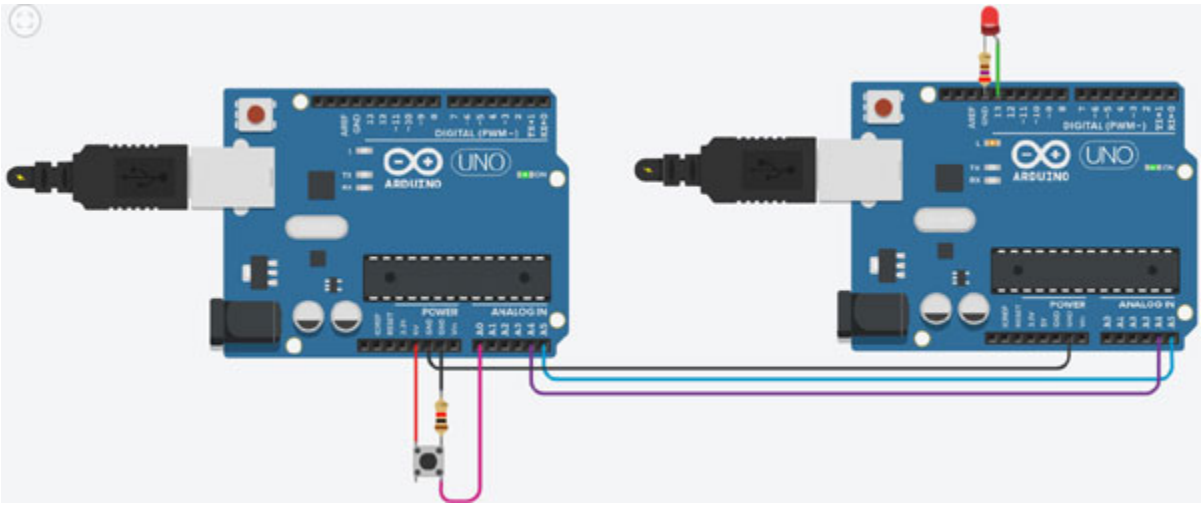


Figure 7.3: *Circuit connections for controlling an LED on slave Arduino with a pushbutton on master Arduino*

The circuit shown in *Figure 7.3* shows the connections between two different Arduino boards. The board on the left is the master board. We have connected the analog pins A4 and A5 of both the boards to each other. This ensures that we have established the hardware connection for the I2C protocol. One important connection is the connection of the ground pins of both boards. If we have more than two devices, we must ensure that all the devices' grounds are connected. If we ignore this, the device will not function properly, since it will not have a common reference voltage. We should also ensure that all the devices' SDA pins are connected and that SCL pins are wired together.

In the master Arduino board, we have connected a pushbutton. Whenever it is pressed, it will send 5V to pin A0; else, the voltage remains 0V. On the slave Arduino, we connected an LED with a resistor at pin 13. Let us understand the code for the master Arduino first.

```
#include <Wire.h>
int button_val = 0;
```

```
\\ In the setup function, we start the I2C communication and
set A0 as the input pin.
void setup()
{
 pinMode(A0, INPUT);
 Wire.begin();
}
\\ In the loop function, we use the transmission function. We
will read the value of the push button and send it. The
process will repeat after a certain delay.
void loop()
{
   Wire.beginTransmission(1);
   button_val = digitalRead(A0);
   Wire.write(button_val);
   Wire.endTransmission();
   delay(200);
}
```

Now, let us understand the code for the slave Arduino:

```
#include <Wire.h>
int button_rcvd =0;
\\ In the setup function, we start the I2C communication, set
pin 13 as the output pin and declare that the function
act_on_recieve is called on receiving the I2C signals.
void setup()
{
 Wire.begin(1);
 Wire.onReceive(act_on_recieve);
 pinMode(13, OUTPUT);
}
\\ In the loop function, we do not mention anything as the
main tasks are performed in the act_on_recieve function.
void loop()
{
```

```
 delay(100);
}
\\ In the act_on_recieve function, we read the received value
and use a simple if-else statement to control the LED.
void act_on_recieve (int howMany)
{
 button_rcvd = Wire.read();
 if (button_rcvd == 1)
 {
  digitalWrite(13,HIGH);
 }
 else
 {
  digitalWrite(13,LOW);
 }
}
```

Recipe 2: Connecting I2C adapter to Arduino

We have learnt that I2C communication provides efficient communication between two devices with only 2 wires. In the previous recipe, we discussed the connections of two Arduino boards and the control of one board with the other. There are a lot of other devices, which support I2C communication and can be connected to other I2C-supported devices. Some of these devices are shown in *Figure 7.4*. Some of the popular I2C-supported devices are as follows:

- 4 Channel 3.3V to 5V Bi-Directional Logic Level Converter
- 128x32 OLED Display Module
- 2.44 cm (0.96 Inch) I2C/IIC 128x64 OLED Display
- 4 channel ADC Module
- I2C Color Sensor
- Gravity O3 Sensor
- Contact-less Infrared Temperature Sensor

- Barometric Pressure and Altitude Sensor

Figure 7.4: *Different devices supporting the I2C communication*

Every device in *Figure 7.4* contains 4 pins. These pins are dedicated to Ground, VCC, SDA, and SCL. Some of these devices have additional pins as well. These devices are specifically designed to support I2C communication. There are a lot of devices that are not supported for I2C communication. If you want to control such devices with I2C communication, you can use an I2C adapter. These adapters will connect to only two I2C pins on the master device and can interface multiple pins on the slave device. These adapters are very specific to the components and are designed for these on a customized basis.

Figure 7.5 shows one such adapter designed for a 16x2 LCD. In the same figure, we have also shown the connections of this I2C adapter with an Arduino board:

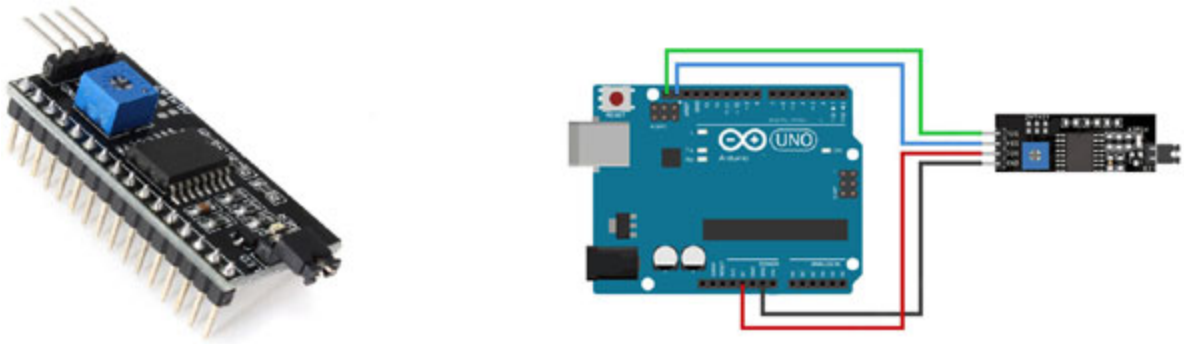


Figure 7.5: *An 12C adapter and its connection with the Arduino board*

Recipe 3: Controlling a 16x2 LCD with I2C protocol

In the previous recipe, we discussed various devices which can be controlled using I2C communication through an Arduino board. We will control a 16X2 LCD with the Arduino board through the I2C communication protocol in this recipe. As discussed, we will connect pins A4 and A5 to the SDA, and SCL pins of the LCD and ground and VCC connections will also be provided. *Figure 7.6* shows the connections of a 16X2 LCD with an Arduino board for I2C communication.

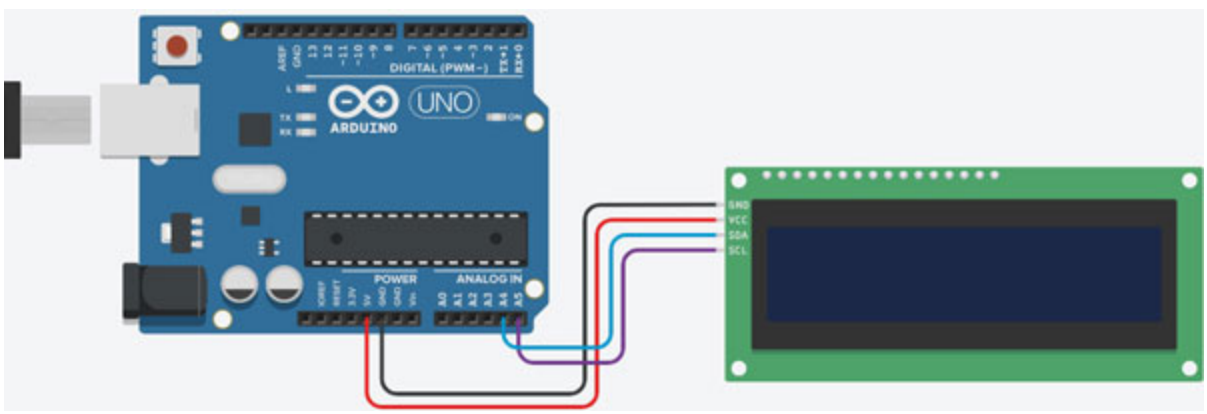


Figure 7.6: *Connecting an I2C-supported LCD with an Arduino board*

We had discussed that for I2C communication, we need to call the **`Wire.h`** library. Since we are using a new LCD, we will also require a new library for the LCD. Therefore, we need to have a few preliminary steps before we can control the LCD through the Arduino board. The first step is to connect the LCD with the Arduino board as per the connections mentioned in *Figure 7.6*. Once the connections are made, we will run a program to

identify the address of the connected I2C device. In recipe one of this chapter, we had mentioned that for I2C protocol, we have to first identify the address and then we can send the data. Connect your Arduino board with the I2C LCD and then run the following program. Open the serial monitor and check the address that is being sent by the device. The sketch is also available at the following link:

https://playground.arduino.cc/Main/I2cScanner/. The same code is written below.

```
#include <Wire.h>
void setup()
{
Wire.begin();
Serial.begin(9600);
Serial.println("\nI2C Scanner");
}
void loop()
{
byte error, address;
int Devices;
Serial.println("Scanning…");
Devices = 0;
for(address = 1; address < 127; address++ )
{
Wire.beginTransmission(address);
error = Wire.endTransmission();
if (error == 0)
{
Serial.print("I2C device found at address 0x");
if (address<16)
Serial.print("0");
Serial.print(address,HEX);
Serial.println(" !");
Devices++;
}
```

```
else if (error==4)
{
Serial.print("Unknown error at address 0x");
if (address<16)
Serial.print("0");
Serial.println(address,HEX);
}
}
if (Devices == 0)
Serial.println("No I2C devices found\n");
else
Serial.println("done\n");
delay(5000);
}
```

The output of this sketch using the LCD was as follows:

```
Scanning…
I2C device found at address 0x3F !
done
```

It implies that the address of my LCD device is 0x3F. It can be different for your devices, and it is necessary to check this address before moving ahead.

The next important task is to add the related libraries to support an I2C-enabled LCD by following the given steps:

1. Go to the link **https://bit.ly/I2C_LIBRARY** and download the .zip file.
2. Extract the contents of the .zip file; it should be a folder named **`LiquidCrystal_I2C`**.
3. Copy this folder to the libraries folder of your Arduino installation directory. By default, it can be **`C:\Program Files (x86)\Arduino\libraries`** for a Windows-based PC.
4. Now, the libraries will be included in your library manager.
5. You can also go to **`Sketch`** -> **`Include library`** -> **`Add a .zip file`** to include this library in your program.

Now, we have come to the last and most important part of coding. Let us write the sketch to print "**`Hello World`**" on the LCD. First of all, we include the required libraries and set the address of the LCD at 0x3F (it will be different for different devices):

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x3F, 16, 2);
```

In the setup function, we begin the LCD module, turn on the backlight, and print "**`Hello World`**":

```
void setup()
{
  lcd.begin();
  lcd.backlight();
  lcd.print("Hello, world!");
}
```

We do not perform any task in the loop function:

```
void loop()
{ }
```

Therefore, we can run a simple sketch using the I2C protocol on a 16x2 LCD. *Figure 7.7* shows the actual setup displaying the same output on an LCD:

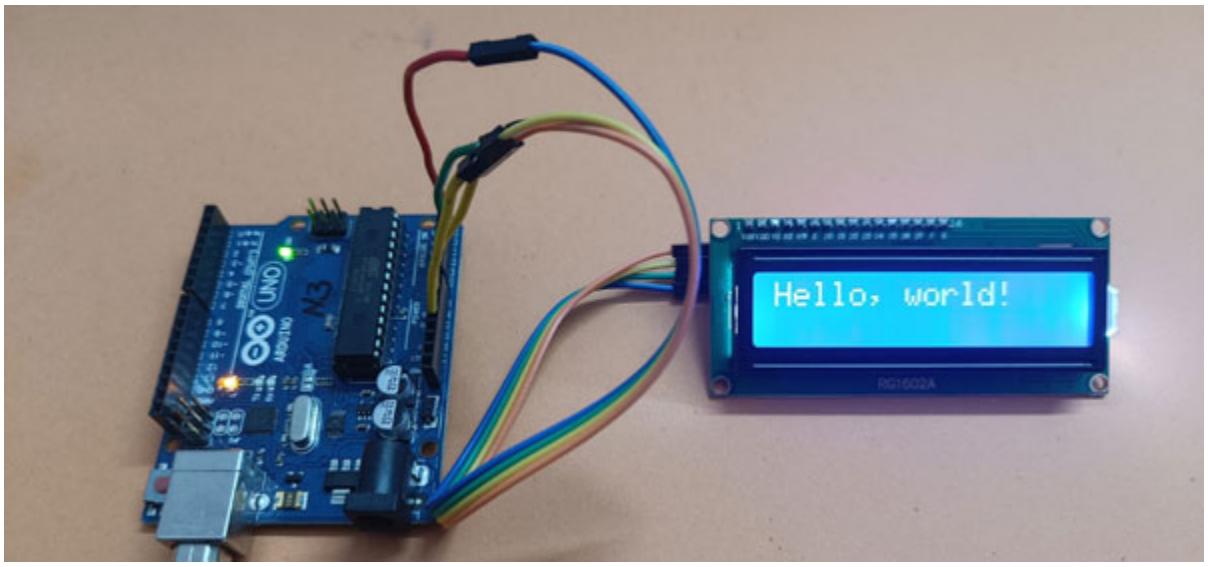


Figure 7.7: *Displaying Hello World on an LCD through I2C communication*

Recipe 4: Displaying data on an OLED screen

We have studied the LCD in the previous recipe; now, let us explore the **Light Emitting Diode** (**LED**)-based **organic light-emitting diodes** (**OLED**) displays. We have these display units in almost every modern device, like a TV, mobile phone and laptop. These display units can display a much larger number of pixels on a smaller-sized screen. In real-world applications, these display units can display millions of pixels on a small smartphone screen and even larger numbers on laptops.

In this recipe, we will use one such OLED screen and learn how to display data on the same. We are using a 128x64 OLED screen. This implies that we can display 128 pixels in a row, and there are 64 such rows. We can display text, lines, shapes and even animations on these displays. We can also use different colors to display the message. A 128x64 OLED supporting I2C communication has 4 pins. These are the same pin names that we had in the I2C-supported LCD. Therefore, the connections will be quite like in the last recipe. *Figure 7.8* shows an OLED connected to an Arduino board with 4 wires and displaying a message. Refer to the following figure:

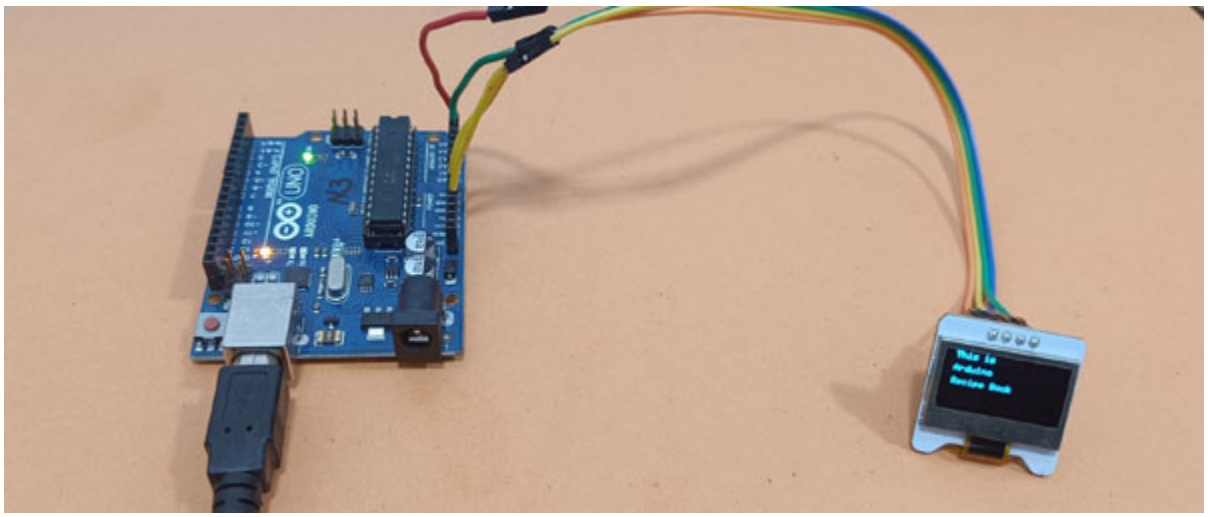


Figure 7.8: *Displaying data on the OLED screen*

Now, let us discuss the coding for the recipe. The important part of the coding of any new component is library inclusion. We must identify the device's manufacturer and install the relevant libraries for the device. The OLED screen used in this recipe is manufactured by the Adafruit company.

So, the first task is to identify and install the relevant libraries. Refer to recipe named *Adding and accessing a new library in Arduino sketch* of *chapter 1, Installing Arduino IDE*, and install the following libraries via library manager:

- **`Adafruit_GFX`**
- **`Adafruit_SSD1306`**

Follow the given process to run a sample code on the OLED screen:

1. Connect the 4 pins of OLED (VCC, Ground, SDA and SCL).
2. Run the code used in Recipe 3 to know the address of the OLED.
3. In Arduino IDE, ensure that both libraries are installed.
4. Open IDE and go to **`File`**-> **`Examples`** -> **`Adafruit SSD1306`** -> **`ssd1306_128x64_i2c`** (as shown in *Figure 7.9*).
5. The sketch will open in the IDE.
6. Compile and upload the sketch.
7. Enjoy the animations on the OLED screen.

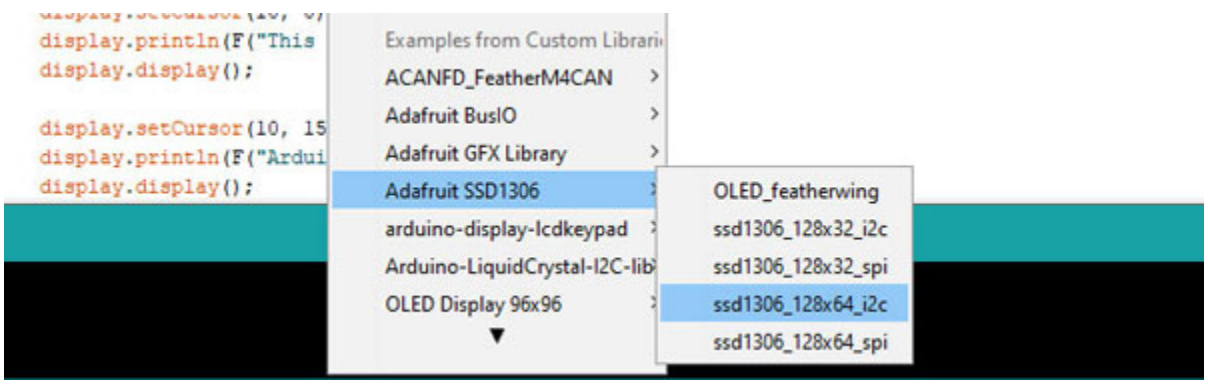


Figure 7.9: *Path to the sample program for a 128x64 OLED screen*

We can use the same process even if we purchase any other size of OLED. The same library will cover them. We must select the size of the OLED carefully. The next task is to display something of your choice. We should learn some of the basic functions of the library, and we can write anything of our choice on the screen. Some of the useful functions are mentioned in *Table 7.1*:

Function name	Explanation
`display.clearDisplay()`	Clear the display and turn off all pixels
`display.setCursor(x,y)`	Set the position of the first character to be displayed
`display.setTextSize(n)`	Size 1 is the smallest and set as default; we can use any size till 8 for larger texts
`display.print("your message")`	Display the String "your message" similar to the Serial.print function
`display.drawPixel(x,y, color)`	Plot a pixel at x,y position in the given color
`display.display()`	Call this method for the changes to take effect

Table 7.1: *Commonly used functions for an OLED screen*

Let us write the code to print "**`This is Arduino Recipe Book.`**" Initially, we declare the libraries and mention the size of the screen. We attach the OLED to the given dimensions:

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels
#define OLED_RESET -1 // Reset pin # (or -1 if sharing Arduino
reset pin)
#define SCREEN_ADDRESS 0x3C ///< See datasheet for Address;
0x3D for 128x64, 0x3C for 128x32
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire,
OLED_RESET);
```

In the setup function, we use the OLED functions. We have used these functions and displayed the message in 3 rows. We have set three different positions, that is, (10,0), (10,15) and (10,30), and then we display three strings to write the full message:

```
void setup() {
  if(!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
    Serial.println(F("SSD1306 allocation failed"));
    for(;;); // Don't proceed, loop forever
```

```
  }
  display.clearDisplay();
  display.setTextSize(1); // Draw 2X-scale text
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(10, 0);
  display.println(F("This is"));
  display.display(); // Show initial text
  display.setCursor(10, 15);
  display.println(F("Arduino"));
  display.display();
  display.setCursor(10, 30);
  display.println(F("Recipe Book"));
  display.display();
  delay(4000);
  // display.clearDisplay();
  // display.display();
  // delay(100);
}
```

The **`loop()`** function is blank. We could have written the same display functions in the loop function as well if we wanted them to display in a loop:

```
void loop() {
}
```

Recipe 5: Digital voltmeter design using Arduino

In this recipe, we will explore a new I2C-supported display device. The device is a 4-digit 7-segment display. When we discussed 7-segment display in recipe *Driving a 7-segment LED display* of chapter *Switches and Displays*, then we used several wires to drive a single display unit. Now, we are trying to display the output on 4 different 7-segment displays. If we use the previous approach, we can never have the required number of wires connected on an Arduino board. Therefore, we use an I2C-supported backpack unit. The unit is described at **https://learn.adafruit.com/adafruit-led-backpack/0-dot-56-seven-**

segment-backpack. *Figure 7.10* shows one such 7-segment backpack. The front and back view of this backpack are shown in the figure. We can observe that the whole unit is comparable to a coin in size. Please refer to the following figure:

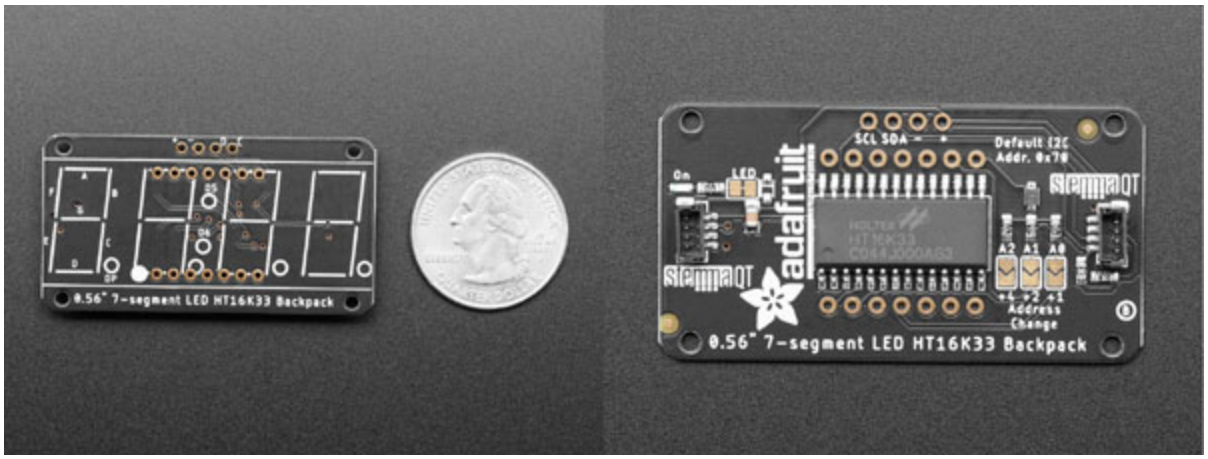


Figure 7.10: *The front and back sides of the 7-segment backpack*

We can observe that there are four separate 7-segment displays on the front, and it requires only 4 pins for connections. Using this backpack, we will make a digital voltmeter. A digital voltmeter is a device that will display the voltage at any point. We will display the voltage up to two decimal points. In this recipe, we have taken a 30 volts power supply and provided different voltage values using the same. There are a few hardware related points that we should consider before connecting the wires.

We are providing the input voltage in the range of 0 to 30 volts and expect the displayed output to be in the same range. However, any pin of Arduino cannot take such a high voltage; therefore, we use a voltage divider circuit to convert the 0 to 30 volts range into a 0 to 5 Volts scale.

Figure 7.11 shows the voltage divider circuit made using two resistors, and the output is connected at analog pin A1. We connected the 7-segment backpack to the I2C pins in the same circuit to display the output.

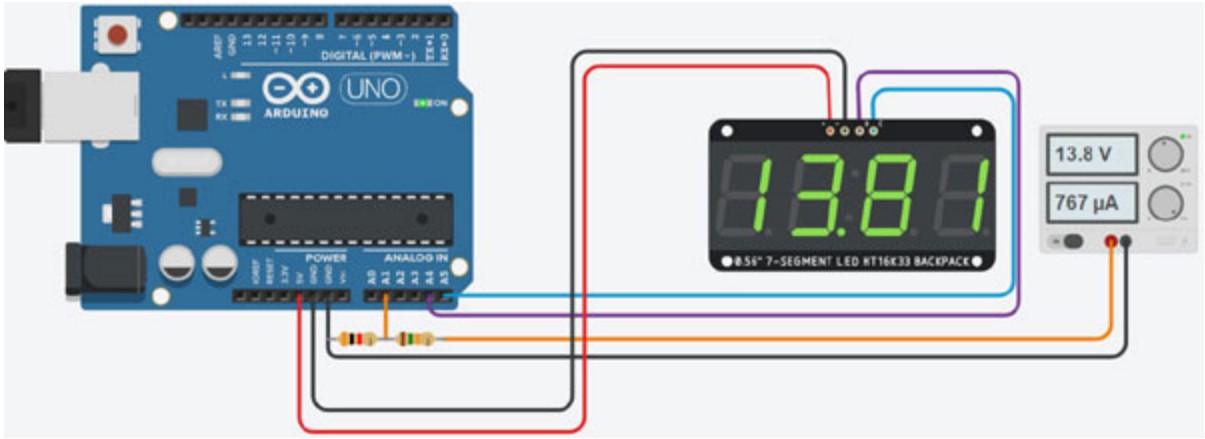


Figure 7.11: *Connections of a voltage source and 7-segment backpack for digital voltmeter recipe*

Here, we are using a new device, so it is necessary to check the required libraries installed in the IDE. For this 7-segment backpack, we need two libraries:

- **`Adafruit_LEDBackpack`**
- **`Adafruit_GFX`**

Follow the given process to run a sample code on the 7-segment backpack:

1. Connect the 4 pins of the backpack (VCC, Ground, SDA and SCL).
2. Run the code used in Recipe 3 to know the address of the backpack. In this recipe, we got the address 0x70.
3. In Arduino IDE, ensure that both libraries are installed.
4. Open IDE and go to **`File`**-> **`Examples`** -> **`Adafruit LED Backpack library`** -> **`sevenseg`**.
5. The sketch will open in the IDE.
6. Compile and upload the sketch.
7. A counter will run on the LED backpack.

In this recipe, we have connected the output of the voltage divider circuit at Pin A1. The analog values at pin A1 (0 to 1023) should map to voltage (0-30). Therefore, we divide the value received at A1 by 1023 and multiply it by 30. Effectively multiply the value by 30/1023 = 0.0293255. The sketch for the recipe is as follows:

```
#include <Wire.h>
```

```
#include <Adafruit_GFX.h>
#include “Adafruit_LEDBackpack.h”
Adafruit_7segment led_display1 = Adafruit_7segment();
void setup() {
   led_display1.begin(0x70);
   pinMode(A1, INPUT);
}
void loop()
{
  // Convert from 0-1023 range to 0-30V range
  // (.0293255 = 30.0 / 1023)
  led_display1.println((analogRead(A1) * 0.0293255));
  led_display1.writeDisplay();
  delay(100); // Wait for 100 millisecond(s)
}
```

Recipe 6: Understanding SPI communication on an Arduino board

In this chapter, we have discussed the I2C communication protocol. I2C is a serial communication protocol that enables the communication between multiple I2C-supported devices using only 4 wires. Now, we will explore one more serial communication protocol: **Serial Peripheral Interface** (**SPI**). SPI was proposed by Motorola company in 1970. SPI is superior to I2C as it is a full-duplex connection. It implies that we can have two-way communication at a given time. We can send and receive the data simultaneously.

SPI communication is used for various peripheral devices, which can be attached to the microcontrollers supporting this communication protocol. SPI is a better protocol for long-distance communication as compared to I2C. Even two microcontrollers can be connected to each other using the same protocol. We need 4 wires apart from Ground and VCC to completely connect and control an SPI-supported device. On Arduino UNO, we have digital pins 10, 11, 12 and 13 dedicated for SPI communication.

Table 7.2 shows the PINs and their functions on Arduino Uno. These pin locations can be different on different Arduino boards, but the functionality will remain the same. So, if you are using another board, you must identify the pins for SPI accurately and connect them accordingly. Let us understand these pins and their functions in detail now:

Pin Number	Pin Name	The function of the Pin
10	SS (Select Slave)	The master uses this pin to enable or disable a particular slave device.
11	MOSI (Master Out Slave In)	This pin sends the message from the master to the peripheral devices.
12	MISO (Master in Slave Out)	From this line, the master will receive the data from the slave devices.
13	SCK/SCLK (Serial Clock)	This pin is used to synchronize the clock signals between different devices.

Table 7.2: *SPI pins and their functions on Arduino UNO*

Figure 7.12 shows the connections of two SPI-supported devices. In this figure, we can observe that the MOSI pin of the master is connected to the MOSI of the slave, and the MISO of the master is connected to the MISO of the slave. The directions of the signal flow are also mentioned. It clarifies which device will be the master and which ones will be the slave. The major difference between the master and slave will be visible at pins SS and SCLK. There are no reverse connections for these signals from the slave to the master. Therefore, only the master can control the other devices by enabling or disabling certain slaves and synchronizing the clock with its own clock signal.

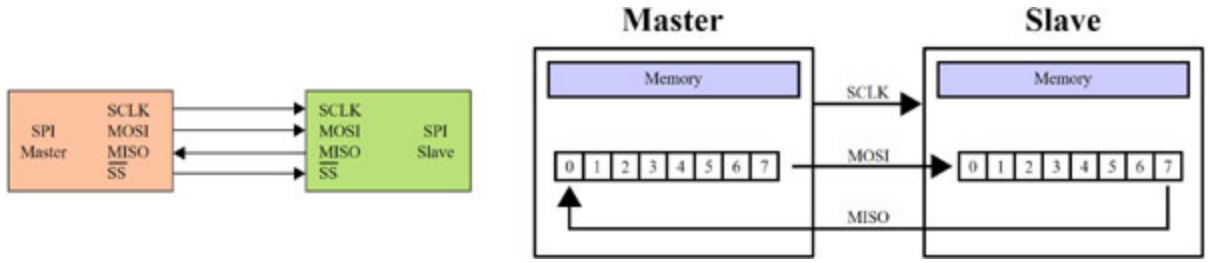


Figure 7.12: *The connections and internal configuration for an SPI communication*

Now, let us understand the basic requirements to control any other device using Arduino through SPI communication. We need the **`SPI.h`** library to access all the functions related to the SPI communication. The library is built into the library manager, and we should just call it in the program.

Some of the common functions in this library are as follows:

- **`SPI.begin()`:** This function is used to start the SPI communication, and it will initialize the SPI bus. SCK, MOSI, and SS are set as outputs in the initial setup. We set SCK and MOSI low and SS high as default values.
- **`SPI.setClockDivider`(divider):** This function sets the different clock frequencies for different slave devices. This is optional, but we may require it in specific cases. The clock frequency will be divided with respect to the master clock frequency. The available dividers are 2, 4, 8, 16, 32, 64 or 128. For example, we can use **`SPI_CLOCK_DIV2`** to divide the clock by 2 or **`SPI_CLOCK_DIV64`** to divide by 64.
- **`SPI.attachInterrupt`(handler):** This function is called when a slave device receives data from the master.
- **`SPI.transfer`(val):** This function is used when we need to send and receive data. This is the main difference between the SPI and I2C protocols.

Recipe 7: Communicate between two Arduino boards using SPI communication

In the previous recipe, we understood the process and requirements for setting up SPI communication. In this recipe, we will demonstrate the same by connecting two Arduino boards with each other and simultaneously communicating from one to another. In this recipe, we need two Arduino boards to be connected to the computer simultaneously.

The circuit connections of the boards are shown in *Figure 7.13*:

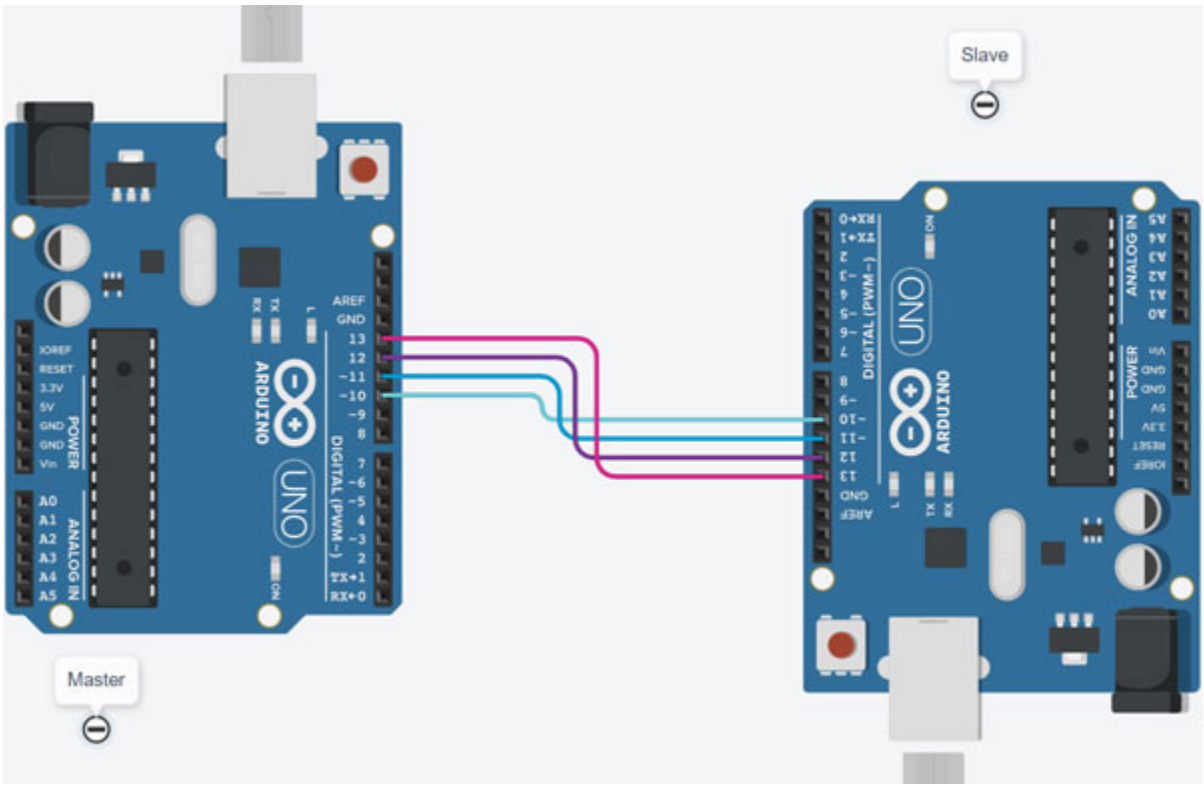


Figure 7.13: *Connections of two Arduino boards for SPI communication*

The connections are simple and have already been explained in the previous recipe. We should connect the two boards' SS, SCLK, MOSI and MISO pins. We decide that the board on the left side is the master, and the one on the right is the slave.

Now, we should connect the master Arduino board to the computer and carefully note the COM port. In my case, the PORT for the master Arduino board is COM3. So, we must ensure that whenever we want to upload the code on master, we must select COM3. The sketch for the master board is as follows:

```
#include <SPI.h>
const int slaveSelectPin = 10;
void setup() {
  pinMode(slaveSelectPin, OUTPUT);
  SPI.begin();
  Serial.begin(9600);
}
void loop()
```

```
{
  digitalWrite(slaveSelectPin, LOW);
  delay(100);
  //Now, the master will send character A, whose ascii value is
  65
  char test = SPI.transfer(65);
  delay(2000);
  // Now, we set the SS to HIGH to disable the slave
  digitalWrite(slaveSelectPin, LOW);
  Serial.println(test);
}
```

In this sketch, we send the ASCII value of character **A**, which is 65, through the master board. Now, on the slave board, we will receive the same. If the received signal is correctly identified, we will send an acknowledgement signal **F**. Therefore, both master and slave will send the data simultaneously.

Now, connect the slave board to the computer. The master board is already connected. When you go to **Tools -> port** and select the correct port for the slave board; in this example, COM4 is the port for the slave board. *Figure 7.14* shows the selection of the slave port when two Arduino boards are connected to the computer. Refer to the following figure:

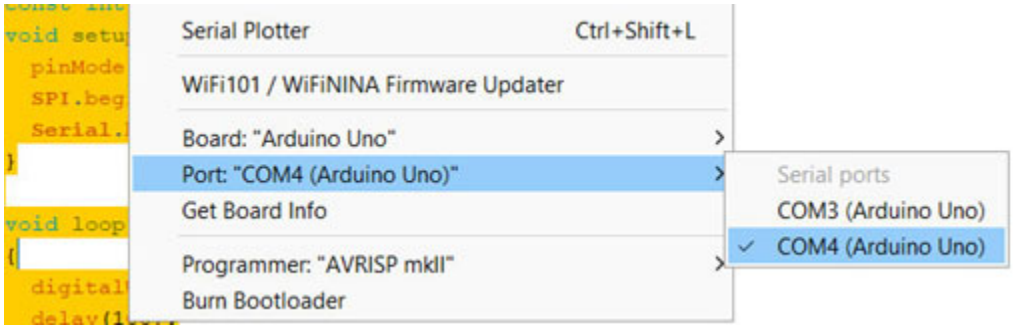


Figure 7.14: *Selecting the slave board on Arduino IDE*

The sketch for the slave is as follows:

```
#include <SPI.h>
bool flag = 0;
byte s_recieve;
```

```
void setup() {
 pinMode(MISO, OUTPUT);
 Serial.begin(9600);
 SPCR = (1 << SPE)|(1<< SPIE);
 sei();
}
void loop()
{
 Serial.println((char(s_recieve)));
 delay(800);
 if(flag)
 {
  SPDR = 'F';
 }
}
ISR (SPI_STC_vect)
{
 s_recieve = SPDR;
 flag = 1;
}
```

In this sketch, we are receiving the incoming signals from the master and storing them in the variable **`s_recieve`**. Then, we convert this value into the character and print it. It will print **A**. Then, we transmit **F** back to the master.

The next step is to view both the serial monitors for COM3 and COM4. In Arduino IDE, we have the limitation that we can view only one at a time. Therefore, we cannot observe both serial monitors at the same time. To resolve this problem, we install Tera Term software. It is free software to print any serial communication data. The software is available at **https://osdn.net/projects/ttssh2/releases/**. *Figure 7.15* shows the landing page, and we should select **`teraterm-4.106.exe`** for downloading.

Figure 7.15: *Download location for tera term executable file*

Once the software is downloaded, follow the steps shown in *Figure 7.16* to install the software:

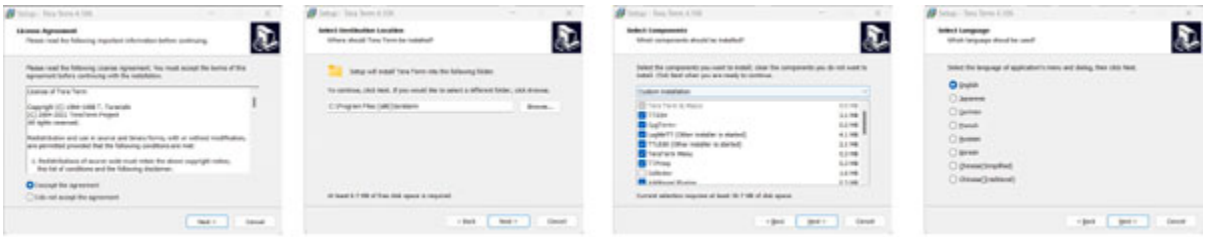


Figure 7.16: *Steps to install the tera term software*

Once tera term software is installed, ensure that both master and slave Arduino boards are connected to the computer and have their respective sketches uploaded. Now, we can open the serial of master (COM3) from IDE and open tera term to open the other serial monitor (as shown in *Figure 7.17*):

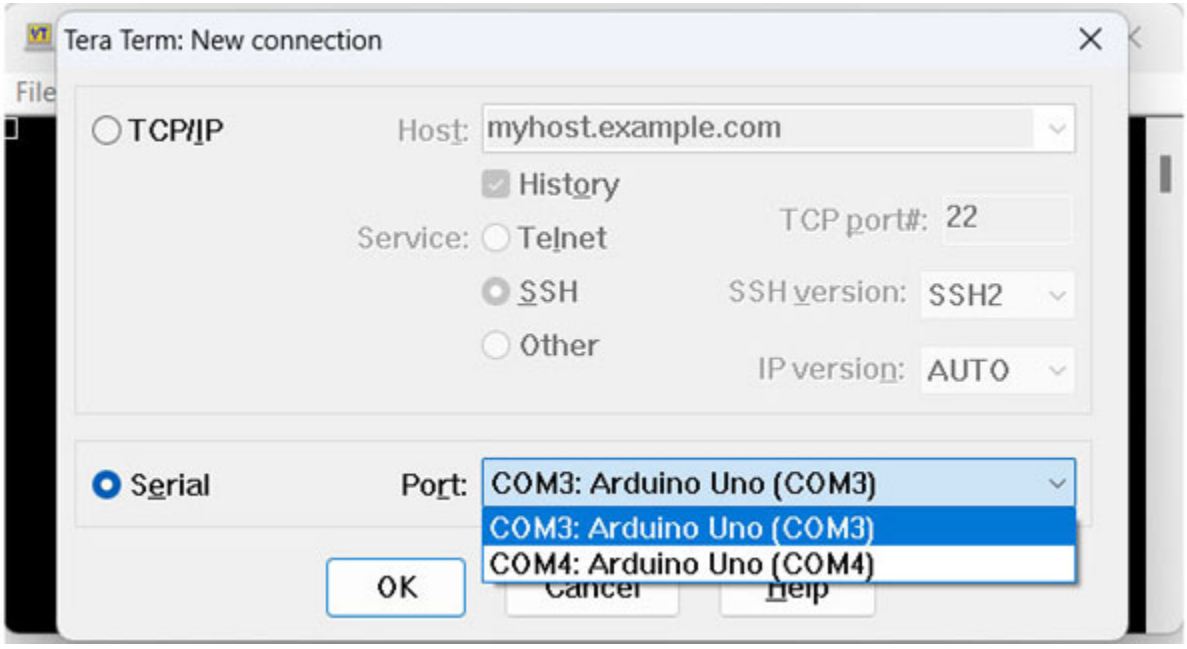


Figure 7.17: *Selecting one port on the tera term software*

Now, both serial monitors are visible on our screen. *Figure 7.18* shows two serial monitors side by side to display the outputs on master and slave Arduino boards. We can observe that the master prints A on the slave, and the slave prints **F** on the master board.

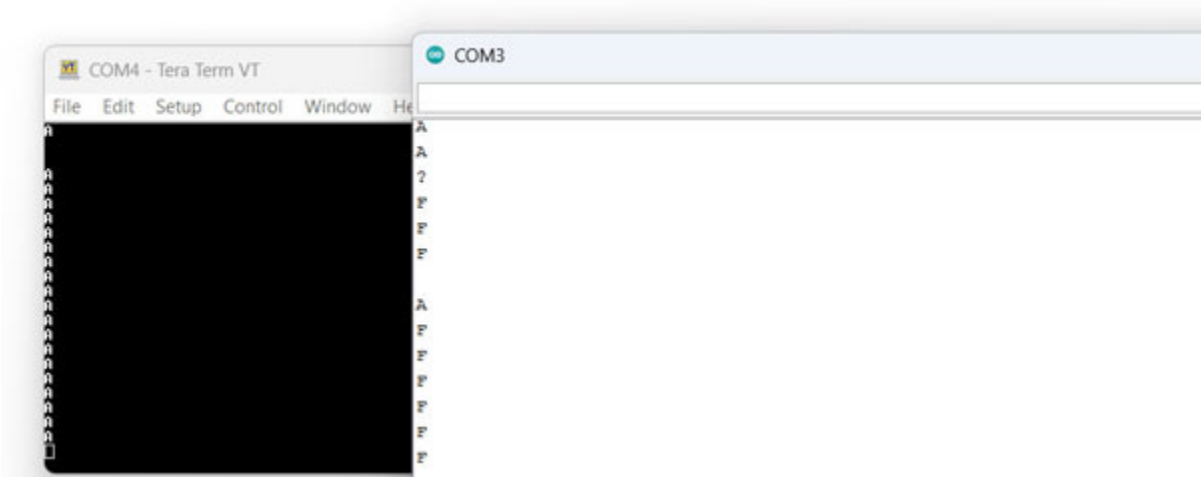


Figure 7.18: *Displaying the master and slave serial monitors for SPI communication*

Conclusion

In this chapter, we started discussing serial communication through the Arduino board. We discussed two in-built serial communication protocols: I2C and SPI. These are the two most popular protocols for serial communication between an Arduino board and peripheral devices. We learned that using the I2C protocol, we need only 2 wires to send data from the master to the slave device. We also designed two interesting recipes using the same protocol. We realized that controlling an LCD takes only 2 wires instead of the 8 wires used in the earlier approach. The I2C communication is one-way communication from master to slave, and we learned SPI communication to overcome this limitation. Using SPI communication, we can send and receive the data simultaneously. This protocol is used in various sensors, to receive the data in real time.

In the next chapter, we will learn another protocol that enables us to connect multiple microcontrollers. We'll learn about the Controller Area Network (CAN) based communication and design interesting recipes using the same.

Points to remember

- An appropriate communication protocol should be selected based on the device.
- The grounds of all the devices should be connected to one single node.
- Ensure that clock pins are connected properly.
- MISO and MOSI should not be interchanged in SPI communication.
- Soldering, if required, should be done carefully.

Multiple choice questions

1. **Which of the following communication protocols supports 2-way communication?**

    a. I2C
    b. SPI
    c. Both
    d. None

2. **Which of the following protocols is developed by Motorola?**

    a. SPI
    b. I2C
    c. Both
    d. None

3. **Which of these protocols is better for long-distance serial communication?**

    a. SPI
    b. I2C
    c. Both
    d. None

4. **The primary function of the SDA pin in I2C communication is to _____.**

a. receive slave data
b. synchronize the clock
c. send data to slave
d. set common ground

5. **The pin that receives the data from the slave to the master in SPI communication is _____.**

   a. SS
   b. SCLK
   c. MOSI
   d. MISO

Answers

1. **b**
2. **a**
3. **a**
4. **c**
5. **d**

CHAPTER 8

CAN Bus Communication

Introduction

In this section, we will discuss the different communication protocols with microcontrollers. In this chapter, we will discuss the **Controller Area Network** (**CAN**). It allows different devices connected to microcontrollers to communicate with each other inside a vehicle. This protocol is particularly designed for smart vehicles. Many security features in cars are controlled using the same communication protocol. In this chapter, we will learn how to use the CAN shield, MCP2515, to connect with an Arduino board and write sketches for the same. We will also transmit the data from a sensor using the same shield.

Structure

In this chapter, we will discuss the following topics:

- Using CAN protocol with Arduino
- Setting up the MCP2515 module with an Arduino
- Sending temperature sensor data over the CAN

Objectives

By the end of this chapter, the reader will be able to do the following:

- Understand the working principle and applications of CAN protocol
- Use CAN protocol module with an Arduino board
- Create a network of 2 and more CAN nodes
- Transmit and receive sensor data over the CAN bus protocol

Recipe 1: Using CAN protocol with Arduino

CAN stands for Controller Area Network. The CAN bus protocol is one the most used and quite robust protocols that allow microcontrollers and devices to communicate with each other's applications without a host computer. This protocol is most widely used in vehicles. Nowadays, vehicles are not made by combining mechanical parts alone. There are more electronics and automation involved in a vehicle.

The CAN protocol enables the communication between different sensors and the actuators of the vehicles without interference from the main microcontroller. Therefore, the main microcontroller is not overloaded with all the computations involved. For example, fuel level, door sensors, odometer, automatic glass wipers, and many more parts operate independently or in coordination with others. Although CAN is widely used in automobiles, there are many other applications of this protocol, such as the following:

- Agricultural equipment
- Electronic equipment for aviation and navigation
- Industrial automation and mechanical control
- Elevators and escalators
- Building automation
- Medical instruments and equipment
- Lighting control systems
- 3D printers

In this recipe, we will understand the working principle of CAN protocol and how to use it using Arduino.

The CAN protocol was initially developed by Robert Bosch and officially released in 1986 at the **Society of Automotive Engineers** (**SAE**) conference in Detroit, Michigan. Intel fabricated the first chip with a CAN controller in 1987, and then other companies also manufactured it. The CAN protocol allows multiple nodes to be connected with each other.

Figure 8.1 shows the connections between three such nodes. In CAN protocol, connecting every note with the central microcontroller is unnecessary. Every node has its own microcontroller and can communicate with the others. By arranging ddifferent nodes at different distances, we can reduce the distance between closely related nodes. One of the biggest advantages of using CAN is

that we don't always have to pass the signal to the central microcontroller. Refer to the following figure:

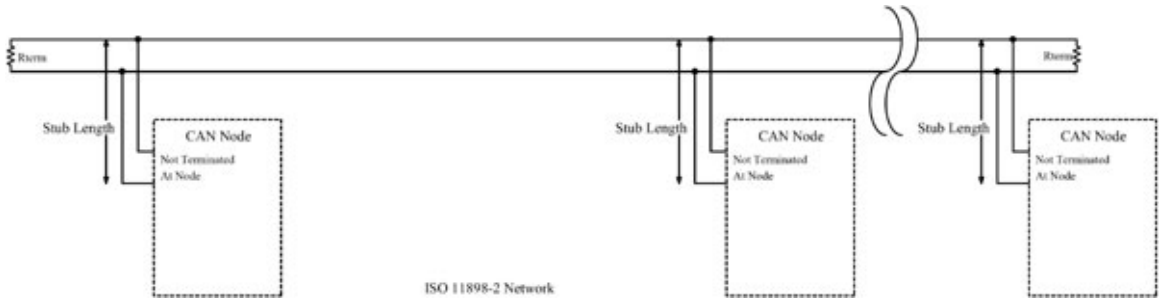


Figure 8.1: *Connections of different CAN nodes with each other*

The CAN protocol consists of two wires for bi-directional data transmission, as shown by two wires in *Figure 8.1*. These two wires are named as follows:

- **`CAN_H`** (CAN High)
- **`CAN_L`** (CAN Low)

These wires act as a differential line. It implies that we can represent either a 0 or 1 by the potential difference between the two wires. When the CAN bus does not carry the signal, the voltage at both lines is 2.5V. The voltage levels will drop and go up to generate the differential voltages.

Figure 8.2 shows the complete CAN frame for data transmission. We can observe that **`CAN_H`** and **`CAN_L`** switch opposite directions to generate the **`CAN_rx`** signal. These differential voltages generate the CAN bitstream of 0s and 1s. Refer to the following figure:

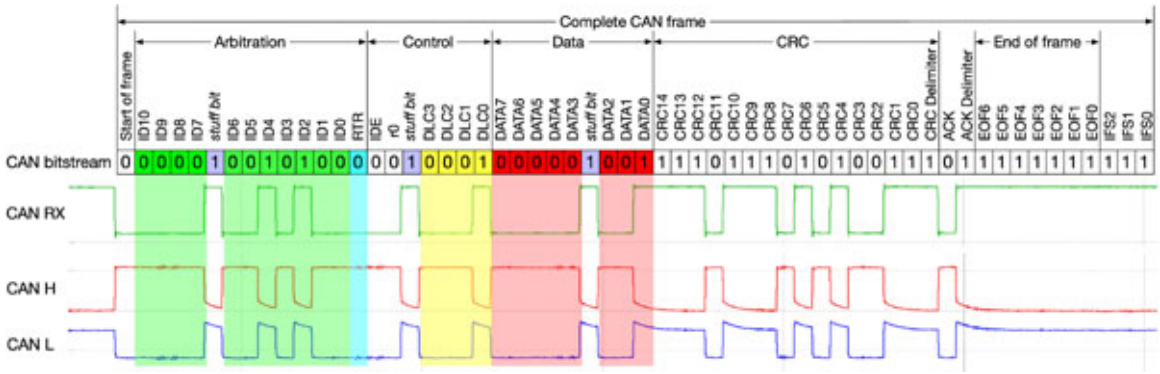


Figure 8.2: *The complete CAN bus frame to transmit data*

In the same frame, we can also observe that data is being sent by using the CAN protocol.

The complete format of the CAN Message is as follows:

Identifier or CAN ID: In *Figure 8.2*, the first set of values are dedicated to the Arbitration. These are also known as CAN IDs or also known as **Parameter Group Number** (**PGN**). These bits are used to identify the CAN devices present in a CAN network. The length of this identifier depends on the version of the CAN protocol. For a standard CAN, the length is 11 bits (ID0 to ID10, as shown in *Figure 8.2*). For an extended CAN protocol, the length is 29 bits.

Data: The data can be sensor data or the control data for the other nodes in the network. The data is sent from one device to another. *Figure 8.2* shows that 4 bits (DLC0 to DLC3) are dedicated to controlling data, and 8 bits (DATA0 to DATA7) are dedicated to the sensor data.

In the next recipe, we'll learn the connections and configurations required to connect this CAN module with the Arduino Uno.

Recipe 2: Setting up the MCP2515 module with an Arduino

We have a good idea about the CAN bus protocol to communicate between different microcontrollers and sensors. CAN chips are designed inside the sensor modules designed for automobiles. However, we can also use generic CAN modules to learn about this protocol and communicate using the same. To connect an Arduino using the CAN protocol, we use a CAN module called MCP2515.

Figure 8.3 shows the top and side views of the module. This is quite a compact (40mmx28mm) rectangular module weighing less than 8 grams. Refer to the following figure:

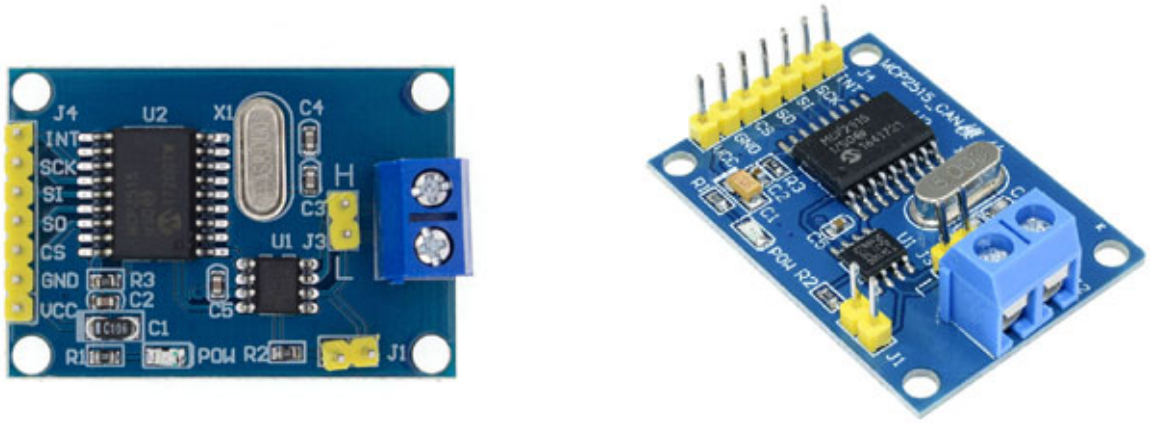


Figure 8.3: *The top and side views of an MCP2515 module*

Some of the salient features of this module are as follows:

- It uses a High-speed CAN transceiver TJA1050.
- This module supports the SPI control for expanding the Multi CAN bus interface.
- It works on an 8MHZ crystal oscillator (mentioned on top of the crystal in *Figure 8.3*).
- The terminal resistance value is 120Ω.
- The module has its independent key, LED, and Power indicators.
- This module supports 1 Mb/s CAN operation.
- In a network, up to 112 such nodes can be connected.

We can observe that there are seven connection pins on one side and two pins on the other side of the board. The name and usage of these pins are as follows:

- **`VCC`:** To provide the 5V supply to the module
- **`GND`:** To connect the ground connection to the module
- **`CS`:** Used as SPI SLAVE select pin
- **`SO`:** The output lead and SPI master input slave
- **`SI`:** The pin is SPI master output slave input lead
- **`SCLK`:** For the SPI communication clock pin
- **`INT`:** Used as an interrupt pin

Two large pins on the other end of the board are the **`CAN_L`** and **`CAN_H`** pins. These are used to connect to the CAN network.

MCP2515 requires a specific set of functions to be operated with an Arduino board. Therefore, we need to embed the required libraries before we start programming this module with Arduino IDE.

The library is available through the library manager. Follow these steps to set the IDE:

1. Go to the Library manager through **`Tools`** -> **`Library manager`**.
2. Search for **`mcp2515`** in the search bar and scroll down to find the auto-wp-mcp2515 library. Install it from the option on the right side. The selection

is highlighted in *Figure 8.4*:

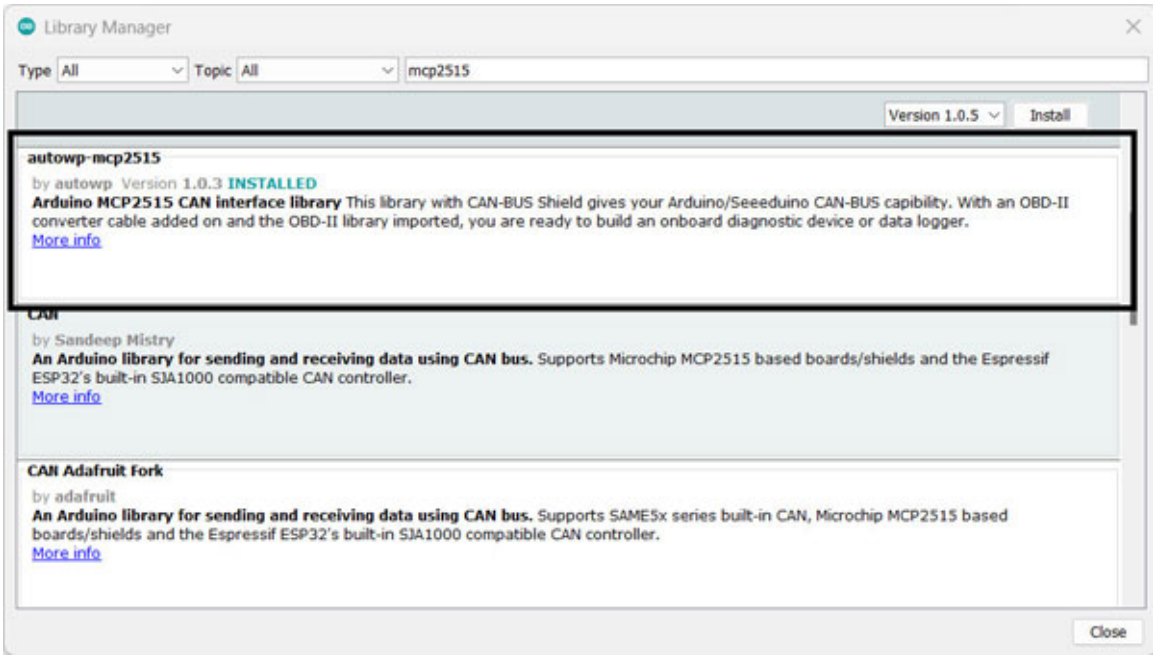


Figure 8.4: *Installing the mcp2515 library from the library manager*

3. The library is installed. We can access the example codes of this library by going to **`File`** -> **`Examples`** ->**`autowp-mcp2515`**. *Figure 8.5* shows the example codes available from this library:

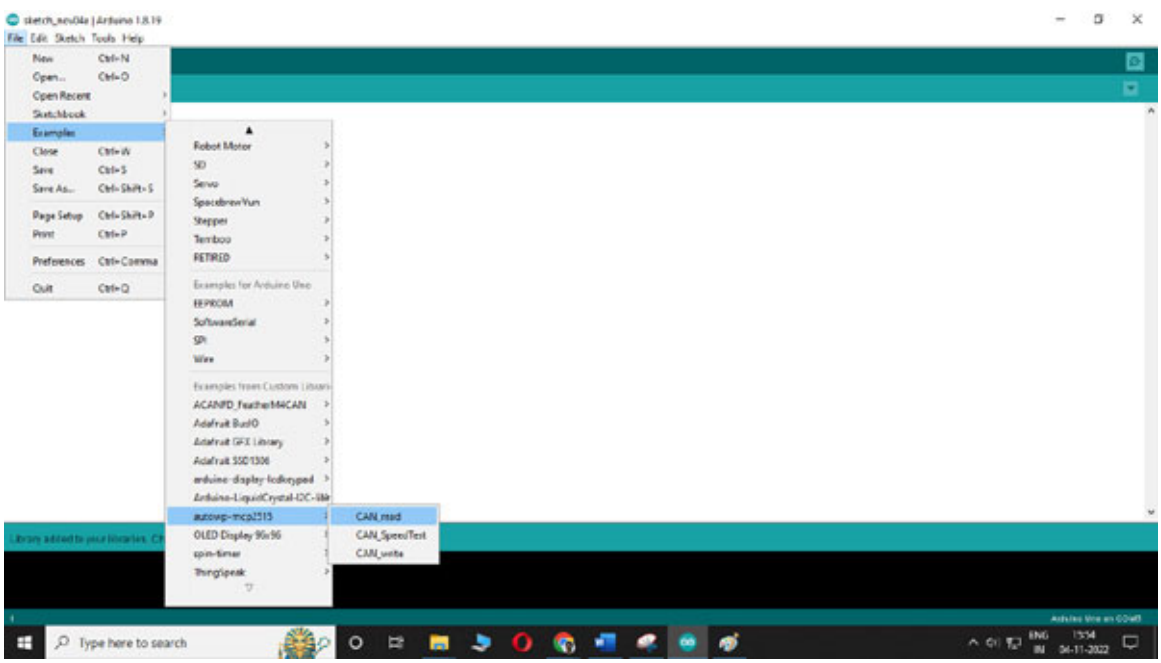


Figure 8.5: *Accessing the example codes from the mcp2515 library*

We can observe that there are codes to read and write the CAN messages. The user can modify these codes to develop their custom applications. Whenever we use CAN-based systems, we need to have a minimum of two CAN modules to set up the complete system. One of the modules will send the data, and the other will receive it through the same protocol.

The next recipe will demonstrate the data transmission and reception using the CAN protocol.

Recipe 3: Send temperature sensor data over the CAN

We have learnt that CAN bus protocol can enable inter-microcontroller communication at a reliable speed. This communication protocol doesn't overburden the main microcontroller of the system. Therefore, it is one of the most preferred protocols for setting up the networks of different sensors and actuators in a vehicle. In this recipe, we will learn the very basic aspect of CAN bus communication between two different CAN nodes of a network. We will create a network of two CAN nodes. The first node will be treated as the transmitter node, and the second one is the receiver node. Will send the temperature values from the transmitter node, receive them on the receiver node, and display them using an LCD.

The connections for the transmitter node are shown in *Figure 8.6*. We can observe that we have not used the interrupt pin on the MCP2515 board. We have attached the ground and VCC from Arduino itself. The other four pins are connected to digital pins 10, 11, 12, and 13. On the board's right side, we connected **`CAN_H`** and **`CAN_L`**. These bins will be connected to the same pins of the receiver board. Refer to the following figure:

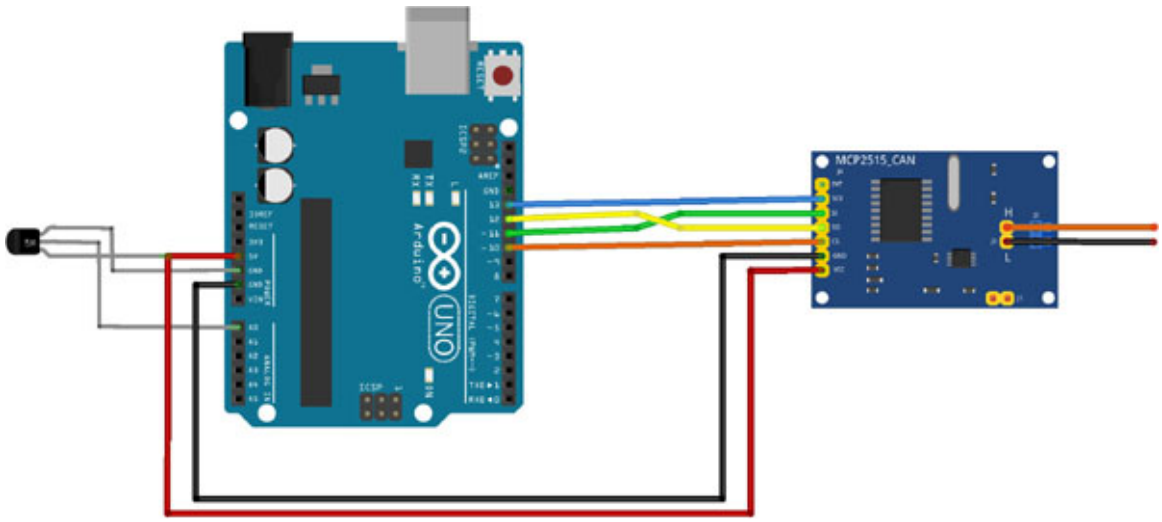


Figure 8.6: *Transmitting the temperature data through the CAN bus*

The code for the transmitter side is written as follows. Let us go through the code and then understand its various components:

```
#include <mcp2515.h>
#include <SPI.h>
#define tempPin A0
int tempValue=0;

struct can_frame canMsg;
MCP2515 mcp2515(10);

void setup()
{
  Serial.begin(9600);
  SPI.begin();
  mcp2515.reset();
  mcp2515.setBitrate(CAN_500KBPS,MCP_8MHZ);
  mcp2515.setNormalMode();
  canMsg.can_id = 0x036;
  canMsg.can_dlc = 1;
}

void loop()
{
  tempValue = analogRead(tempPin);
```

```
  tempValue=(tempValue/1023)*5000;
  tempValue=tempValue/10;
  Serial.print("The value from the temperature sensor is = ");
  Serial.println(tempValue);
  canMsg.data[0] = tempValue; //Update pot value in [0]
  mcp2515.sendMessage(&canMsg); //Sends the CAN message
  Serial.println("Message sent");
  delay(500);
}
```

Let us understand the code in detail now. We have declared the required libraries and a few variables in the following lines. We need **`mcp2515.h`**, which would be available after importing the ZIP file library, as described in Recipe 2 of this chapter. We will communicate using the SPI protocol; therefore, the respective library is also included:

```
#include <mcp2515.h>
#include <SPI.h>
#define tempPin A0
int tempValue=0;
```

Then, we declare a data structure with the name **`can_frame`**, and one component of the **`can_frame`** is **`canMsg`**. This will be the message that we will send. Earlier, we described that the complete CAN frame contains different components (*Figure 8.2*). The message is one of them. We have also mentioned that the chip select pin is connected to the digital pin 10.

```
struct can_frame canMsg;
MCP2515 mcp2515(10);
```

We set up a few protocols and communication-related connections in the setup function:

`Serial.begin(9600)` is used to start the Serial communication at 9600 bps.

`SPI.begin()` starts the SPI communication.

`mcp2515.reset()` is used to reset the MCP2515 module.

`mcp2515.setBitrate(CAN_500KBPS,MCP_8MHZ)` is used to set the CAN at speed 500KBPS and Clock 8MHz.

`mcp2515.setNormalMode()` sets the normal mode for the module.

`canMsg.can_id = 0x036` defines that the unique ID related to this message is 0x36.

`canMsg.can_dlc = 1` declares that the data length is 1 byte of 8 bits.

Now, let us also understand the statements in the **`loop()`** function:

`tempValue = analogRead(tempPin)` reads the temperature value from the tempPin location.

`tempValue=(tempValue/1023)*5000` transforms the ADC value into degree Celsius.

```
tempValue=tempValue/10;
Serial.print("The value from the temperature sensor is = ");
```

`Serial.println(tempValue)` are two lines that print the temperature value on the serial monitor.

`canMsg.data[0] = tempValue;` Now, we assign the temperature value as our CAN message.

`mcp2515.sendMessage(&canMsg)` sends the CAN message.

`Serial.println("Message sent");`

`delay(500);` In the end, we print a confirmation message and wait for 500ms.

Once the transmission node is set, we look at the receiver side. The connections for the receiver are shown in *Figure 8.7*. The connections of the MCP2515 are the same as those of the transmitter node. **`CAN_H`** and **`CAN_L`** are connected to the respective pins of the transmitter node. We have used a 16x2 LCD using the I2C communication protocol. They are connected to pins A4 and A5 of the Arduino board. Refer to the following figure:

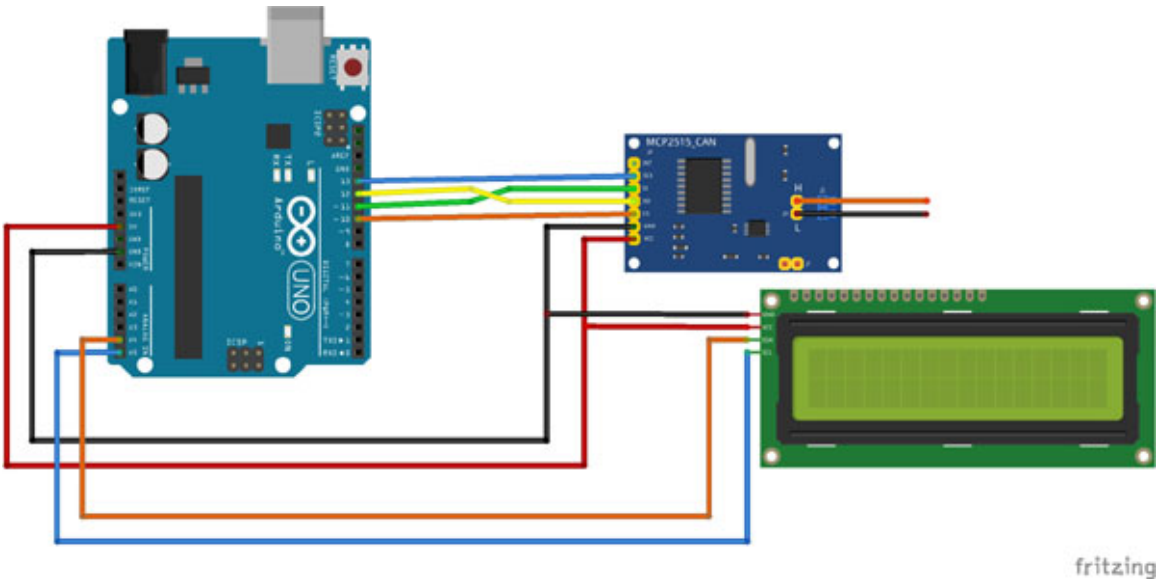


Figure 8.7: *Receiving the CAN bus data and displaying it on the LCD*

The code for the recipe is as follows:

```
#include <mcp2515.h>
#include <SPI.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x3F,16,2);

struct can_frame canMsg;
MCP2515 mcp2515(10);

void setup() {
  lcd.init();
  lcd.backlight();
  delay(1000);
  SPI.begin();
  Serial.begin(9600);
  mcp2515.reset();
  mcp2515.setBitrate(CAN_500KBPS,MCP_8MHZ);
  mcp2515.setNormalMode();
}

void loop()
{
```

```
  if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK)
  {
   if(canMsg.can_id==0x036)
   {
    int x = canMsg.data[0];
    Serial.println(x);
    lcd.setCursor(1,0);
    lcd.print("Temp value …");
    lcd.setCursor(2,1);
    lcd.print(x);
    delay(200);
    }
  }
 }
```

The code is quite straightforward and very similar to the transmitter code. Initially, we declared the required libraries. We also called the I2C LCD library to enable the I2C LCD on the board. In the **`setup()`** function, we defined **`lcd.init()`** to initialize the LCD and **`lcd.backlight()`** to turn on the backlight of the LCD. The remaining statements are similar to the transmitter code. In the **`loop()`** function, we check the CAN message's availability and whether the CAN ID is **`0x36`**. If we satisfy these conditions, then we print the temperature values on the LCD.

This recipe combined the concepts of sensor integration, SPI communication, I2C-based LCD, and CAN bus-enabled communication. In practical systems, we always merge different recipes from this book and design our own custom recipes.

Conclusion

In this chapter, we understood the working principle, components and the operation of a robust communication protocol for vehicles. The CAN bus protocol uses serial data transmission between different CAN nodes without the interference of the central microcontroller. We can assemble multiple nodes in a single network to design such networks. We demonstrated the transmission of the temperature values from one node to the display unit of another node.

The users can always connect multiple sensors to multiple or a single node and transmit/receive the data over the CAN protocol. We must ensure that a unique CAN ID (8-bit hexadecimal code, like 0x36) should be assigned to each data type. Then, any node can send/receive that data by detecting the correct CAN ID.

In the next chapter, we'll start a new section of the book. We'll discuss the wireless connectivity options and their tools and techniques in these chapters.

Points to remember

- CAN ID should be common between the transmitter and receiver.
- The message length should be common for every type of message.
- Connections of four programmable pins of the MCP2515 module should be made with utmost care.
- `CAN_H` and `CAN_L` must be connected to each other.
- The ground of all the power supplies in the system must be connected to one common point.
- Soldering, if required, should be done carefully.

Multiple choice questions

1. **CAN stands for _____.**

    a. Controller Area Network
    b. Controller Air Network
    c. Couple Area network
    d. None of these

2. **The controller area network is a _____ protocol.**

    a. Number-based
    b. Voice-based
    c. Message-based
    d. Code-based

3. **How many pins are present on the left side of an MCP2515 module?**

a. 5
b. 6
c. 7
d. 8

4. **Which pin should carry the high values of the CAN message?**

a. CAN_H
b. CAN_L
c. CS
d. INT

5. **At idle state, the voltage of CAN_L and CAN_H is the same at ____ Volts.**

a. 0
b. 1.25
c. 5
d. 2.5

Answers

1. **a**
2. **c**
3. **c**
4. **a**
5. **d**

CHAPTER 9

Bluetooth Communication With Arduino

Introduction

In the previous unit, we learnt about major serial communication techniques supported by the Arduino board. I2C and SPI are serial communication protocols that need wires connected between two or multiple devices. This chapter starts the last unit of this book, which discusses wireless communication techniques. In this chapter, we will learn to connect and communicate using an HC-06/HC05 module with an Arduino board. We will learn how to connect a smartphone and Arduino through the same network of Bluetooth devices. Once we have a good command over Bluetooth communication, we'll design a few interesting applications using the same.

Structure

This chapter will focus on understanding Bluetooth communication, setting up the hardware and software for it, and designing recipes using Bluetooth communication. The recipes for this chapter are as follows:

- Understanding and establishing a Bluetooth connection with a Bluetooth module
- Configuring your smartphone to connect with Arduino via Bluetooth
- Visualizing serial monitor information over Bluetooth
- Smartphone-controlled RC car
- Bluetooth-controlled servo motor

Objectives

By the end of this chapter, you should be able to do the following:

- Understand the concepts of Bluetooth communication
- Choose appropriate hardware and software for Bluetooth communication
- Transmit data using Bluetooth protocols

- Control two DC motors with commands from a Bluetooth device
- Control the angle of rotation of a servo motor using Bluetooth

Recipe 1: Understanding different Bluetooth modules to connect with an Arduino

Bluetooth is a short-range wireless technology standard that is used for exchanging data between fixed and mobile devices over short distances. The distance can be in the range of a few meters only. Although the distance is small, the Bluetooth protocol is quite reliable and one of the most versatile wireless communication techniques today. We use many Bluetooth-enabled services, like playing music on wireless EarPods, connecting our phone to the car for calls, or simply transferring files from one mobile/PC to another.

Bluetooth is inbuilt in most of these devices, but Arduino does not have inbuilt Bluetooth support. So, how do we communicate using the Bluetooth protocol with an Arduino board? This recipe will help us identify different options for Bluetooth communication with the help of an Arduino board. We use a **Bluetooth Low Energy** (**BLE**) Module and connect it to an Arduino board to establish Bluetooth communication between the board and other devices. The other device can be any other Bluetooth-enabled device like a mobile phone, TV, speaker or even another Bluetooth module.

Types of Bluetooth modules

Bluetooth technologies are an evolving technology, and with time, newer and faster versions are introduced in the market. Currently, the latest version of Bluetooth communication is 5.0. With every new version, communication speed is improved, and security features are enhanced. However, the basic communication protocol remains the same. That implies that a device operating on the latest version can also connect with the Bluetooth devices operating on the older versions. Depending on the version of Bluetooth technology, different modules are available in the market to enable any microcontroller (like Arduino) to communicate via Bluetooth. Some of the popular BLE modules are listed as follows:

Bluetooth 2.0/2.1 EDR Modules:

- HC-06 Bluetooth 2.0 EDR modules

- HC-05 Bluetooth 2.0 EDR modules

Bluetooth 4 / BLE Modules:

- HM-10
- HM-11
- BT05-A mini BLE Bluetooth
- AT-09 Bluetooth V4.0

Figure 9.1 shows all of these Bluetooth modules. All of them are almost equally sized and have 4 or 6 connecting pins. Depending on the type of communication these cards support, we can use a few pins simultaneously. Most of these devices are priced in a similar price range.

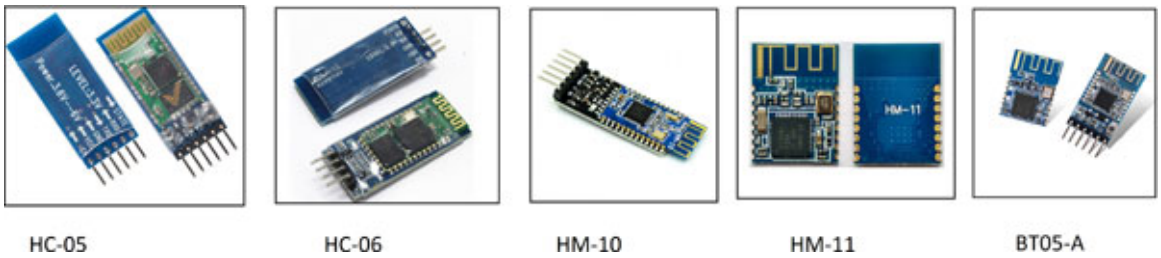


Figure 9.1: *Different Bluetooth modules available to connect with an Arduino board*

Now, we have various modules, and there is an obvious question in front of us: which one should we use in our recipes? Well, to answer that questions, let us first understand different modules in detail and get to know their features and differences from each other. We are picking three modules, i.e., HC-05, HC-06 and HM-10, for comparison. We are covering two different versions of Bluetooth technology in this comparison.

HC-05 is one of the most popular Bluetooth modules that uses Bluetooth 2.0 technology. This module works on both master and slave protocols. It implies that by using this module, we can receive commands from other Bluetooth devices (slave operation) and send commands to others (master operation). It has 6 pins and needs them for full master-slave communication. This module has a BC417 single-chip Bluetooth IC and supports UART and USB interfaces. It can operate from 4V to 6V power supply and works at 2.4GHz frequency. The functions of the 6-onboard pins are explained in *table 9.1*:

Pin name	Function
VCC	Power supply to the module

GND	The ground connection of the module, which should be common with every other ground pin
TXD	Pin that transmits the serial data and is also the output pin of the module
RXD	Pin that receives the serial data on the module
KEY	Pin that switches between the AT command and the default modes; we can set the value for specific operations
State	Indicates whether the module is connected and informs about the state of connectivity of the module

Table 9.1: *Functions of different pins at HC-05 Module*

The AT commands are used to initialize, set parameters, set scan, stop the scan and various other features related to Bluetooth communication. This pin makes this Module one of the most versatile modules and can be used for many applications. In HC-05, there is a small pushbutton to change the AT modes of the module.

HC-06 also works on Bluetooth 2.0 protocols. This module has only 4 pins. Compared to HC-05, it has all the pins except KEY and State. Therefore, this module can work only as a slave module. We cannot use AT commands on HC-06 and use it as a master device. Even though it cannot be used as a master device, it is still preferred in various applications where only slave operations are required. For example, when we want to design a Bluetooth-controlled remote or a car, then we don't need any input from the other device and HC-06 is a great choice for such applications.

CH-06 typically works at 5V but can operate in the range of 3.3V to 6V as well. This module required an operating current of 30mA. The module works at 2.4GHz frequency at a default baud rate of 9600bps. This module does not have the KEY pin; therefore, it is permanently configured in slave mode.

HM-10 supports BLE 4.0. This module is the fastest among the discussed modules. Therefore, it is preferred wherever data transmission is required at a higher speed. The module can work both as a master and slave. The board operates in the voltage ranges of 3.3V to 6V. There are 6 pins on the module. Five pins are similar to HC-05. This board does not have the **`KEY`** pin, and in place of that, HM-10 has a **`BRK`** pin. The **`BRK`** pin is called the **`break`** pin. The **`break`** pin breaks an active connection by bringing it to a LOW value.

Now, we have a good idea about the different programmable modules to enable Bluetooth communication with an Arduino board. In the next recipes, we'll discuss the connections and coding for establishing Bluetooth communication using some of these modules.

Recipe 2: Establishing a Bluetooth connection between a PC and a Bluetooth module

In the previous recipe, we learnt that various Bluetooth modules are available to connect with an Arduino board. In this recipe, we will discuss the connection of HC-06 with a computer. We have chosen HC-06 because it is the simplest module and always works as a slave. The other modules can also be connected in the same manner with slight modifications. The connections are shown in *Figure 9.2*:

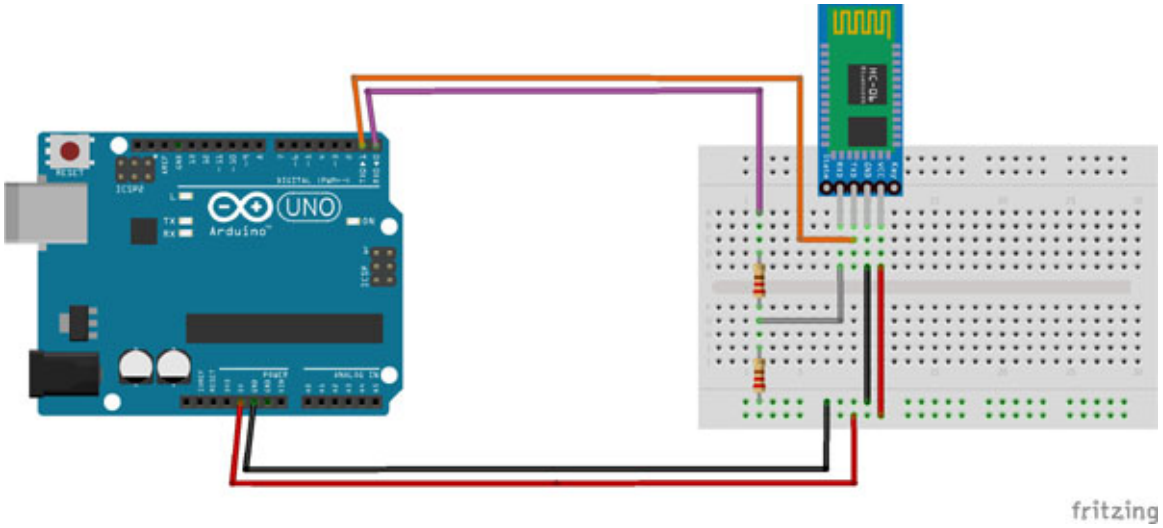


Figure 9.2: *Connecting an HC-06 module to an Arduino UNO board*

We have provided the VCC and ground connections from the Arduino board. On the Arduino board also, we have TXD and RXD pins for transmitting and receiving the serial communication data, respectively. On Arduino Uno, these pins are digital pins 0 and 1. This pin combination is used for any wired serial communication.

The TXD pin is directly connected to the TXD pin from the module to the Arduino board. However, the RXD pin from the Arduino board is not fed directly to the RXD of the Module. We used a voltage divider circuit using two series resistors and then provided the divided voltage to the RXD. This

adjustment is due to the different operating voltages of HC-06 and the Arduino board. The Arduino board provides a 5V output, while the HC-06 module works at 3.3V. Therefore, we used the upper resistor equal to 1kΩ and the lower resistor of 2kΩ to generate 3.3V from 5V.

There are three main points to note while programming any serial communication device with Arduino:

- **For code uploading:** If you are using assigned hardware pins for serial communication, you must detach these pins when uploading the code. This is necessary because the code transfer from the PC to the board is also done as serial communication; therefore, the two protocols must not interfere. So, if you want to upload any code related to Bluetooth communication, then you must remove the connections of PIN 0 and 1 before uploading.
- **For regular operations:** Once the code is uploaded, we must connect the received (RXD) of the module to the transmitter (TXD) of the Arduino board and the transmitter of the Module (TXD) to the received (RXD) of the Arduino board. It makes sense logically as the transmitted data from the Arduino board is received by the module, and the Arduino board receives the transmitted data from the module. We must be very careful while connecting these two pins.
- **For AT commands:** Any Bluetooth module can be checked for its functionality and configured before being used in a project. We can use these commands to set the module's name, password, and other properties. You must have seen that you get a default device name for your new phone or laptop, and then you can change it at per convenience. To connect any module and operate it using AT commands, we should connect RXD to RXD and TXD to the TXD. This pair of connections is opposite to regular operations. *Figure 9.2* shows the Bluetooth module HC-06 connected to the Arduino board for AT mode operations.

Controlling HC-06 with AT commands

Let us first connect and control the HC-06 module using the AT commands. The connections should be made as per *Figure 9.3*:

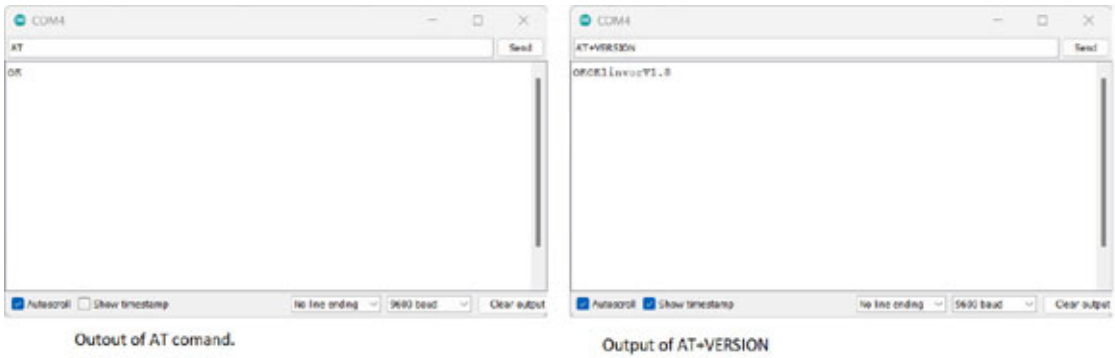


Figure 9.3: *Outputs of AT and AT+VERSION commands from HC-06*

We must ensure that the voltage divider is properly configured to provide 3.3V to the receiver pin of HC-06. We must upload a blank sketch to the board to operate any Bluetooth module in AT mode. **Ensure that you have detached the RXD and TXD pins while uploading.** After detaching the connections, upload the following code:

```
void setup() {
}
void loop() {
}
```

Attach the RXD and TXD wires and open the serial monitor. Now, the serial monitor can show the output for the AT commands. We can use the following AT commands on HC-06:

- **AT**: Check the connection; it will return OK if the connection is established.
- **AT+VERSION:** This will give the version of the Bluetooth module.
- **AT+NAME:** This is used to change the name of the module name; no space between the name and the command.
- **AT+BAUDX:** This is used to change the baud rate. X is the baud rate code. X can be any integer between 1 to 9. where X=1 to 9. 1 set to 1200bps, 2 sets to 2400bps, 3 sets to 4800bps, 4 set to 9600bps (Default), 5 set to 19200bps, 6 sets to 38400bps, 7 sets to 57600bps, 8 sets to 115200bps.
- **AT+PIN:** The PIN is a 4-digit password to connect the HC-06 module. By default, the value is set to 1234. We can change it using the AT+PIN command to any other 4-digit number.

To execute these commands, we should first attach TXD and RXD pins back to the Arduino board. Then, we can execute these commands on the serial monitor. Enter the command and press the **`send`** button. We must ensure the following on the serial monitor settings:

1. Select the baud rate of 9600.
2. Select No line ending in the display settings.

Connecting a Bluetooth module to the Windows PC

Like any Bluetooth device (a headphone, speaker, or printer), we can connect our HC=06 module to the Windows 10 or 11 PC. This will work only on a PC having Bluetooth support. Typically, it is given on laptops and not on desktops. So, you must ensure that your PC has Bluetooth support before you try this recipe. The steps to connect an HC-06 or any other module to a Windows 11 PC are as follows:

1. Connect the HC-06 module to the Arduino or any other source of power.
2. The HC-06 LED will blink, showing that it is ready to pair.
3. Open the settings of the PC.
4. Enable Bluetooth communication on the PC by going to the Bluetooth and devices tab.
5. Click on the **`Add device`** button to add a new device. These steps are shown in *Figure 9.4*:

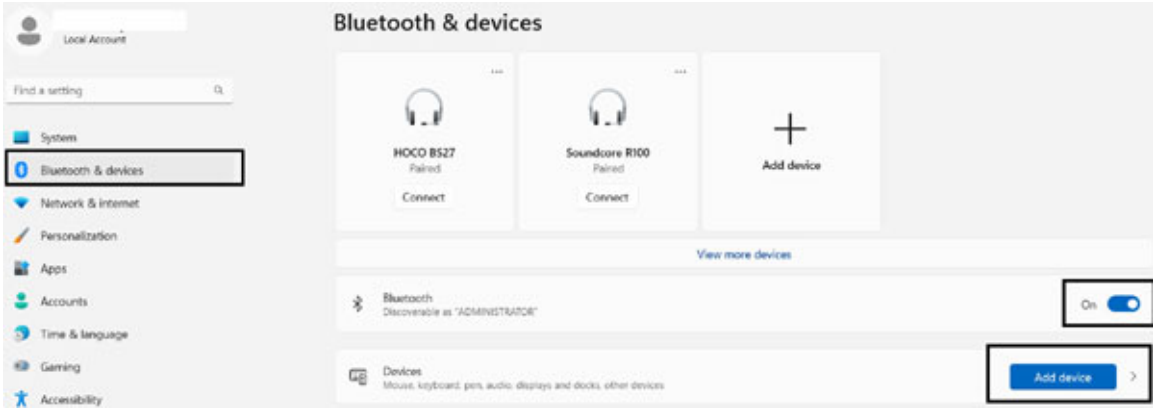


Figure 9.4: *Bluetooth settings on Windows 11 PC and enabling Bluetooth communication*

6. In the add device pop-up, select Bluetooth, as shown in *Figure 9.5*.
7. A list of available devices will appear.

8. Select HC-06 and enter the passcode = 1234 (the default PIN).
9. Click on connect, and the device will be connected to the PC. *Figure 9.5* shows all the steps involved in this process.
10. Similar steps are followed on a Windows 10 PC as well. Only the locations of Bluetooth settings will change as per the GUI of the OS. Refer to the following figure:

Figure 9.5: *Connecting an HC-06 to the windows 11 PC*

Recipe 3: Configure your smartphone to connect with Arduino via Bluetooth

In the first two recipes, we discussed the different Bluetooth modules and their configurations and connections with a PC. In this recipe, we will discuss the configuration of an Android or iOS-based mobile phone with HC-06 or any other Bluetooth module. We will learn how to connect an HC-06 module to a mobile app and send commands to an Arduino board from the mobile phone.

At first, we should install an Arduino supported Bluetooth app on our respective devices. Arduino Bluetooth controller is a reliable app for Android, and Bluetooth for Arduino is a similar app for iOS devices. Both these apps are free to use, and there are a few more similar apps. The links are as follows:

Arduino Bluetooth controller:

https://play.google.com/store/apps/details?id=com.giumig.apps.bluetoothserialmonitor&hl=en_IN&gl=US

Bluetooth for Arduino: **https://apps.apple.com/us/app/bluetooth-for-arduino/id1505096526**

Figure 9.6 shows some steps involved in installing the app for Android OS. Part A shows the play store image, and part B is the interface that we get on

installing the app. The app may require certain permissions, and we should allow them, as shown in Part C.

After installation, follow these steps to connect an HC-06 to the mobile app:

1. Connect the VCC and ground of HC-06 to the 5V and ground of the Arduino UNO. The LED should blink and indicate that the module is active and ready to pair now.
2. Go to your mobile's settings and turn on Bluetooth communication.
3. Scan for Bluetooth devices and pair HC-06. The default passcode is 1234. Part D of *Figure 9.6* shows the detected device with the name HC-06.
4. Now, open the Arduino Bluetooth Controller app and select HC-06. You will get 4 different ways to connect to the module. These are shown in part E of *Figure 9.6*:

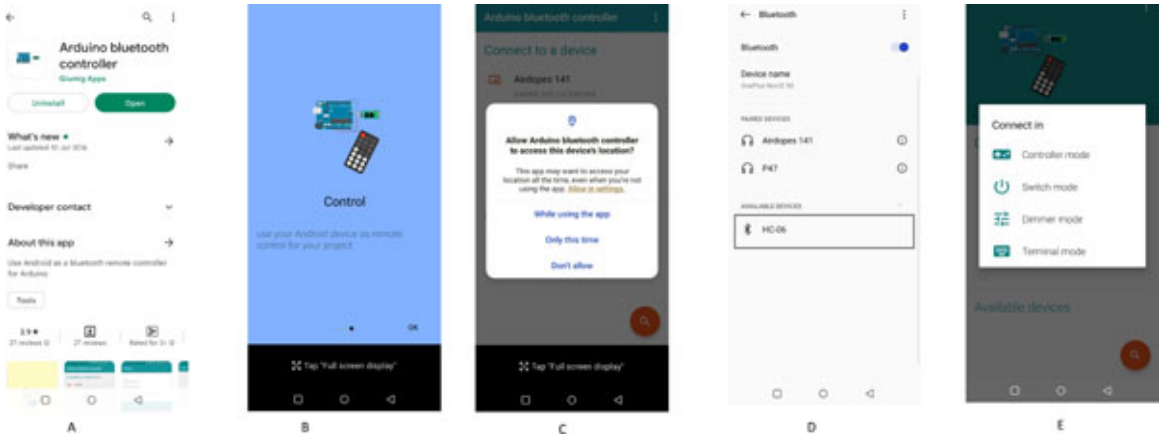


Figure 9.6: *Installing the Bluetooth controller app and pairing the HC-06 module with it*

Following the preceding steps, we have established the connection of the HC-06 module with our mobile phone. Now, the next crucial step is to send and receive signals from mobile phones to the Arduino board through the Bluetooth module. This will require configurations on the mobile app and coding on the Arduino sketch supported by the hardware connections on the Bluetooth module with the Arduino board. Let us understand each of these in detail now. Refer to the following figure:

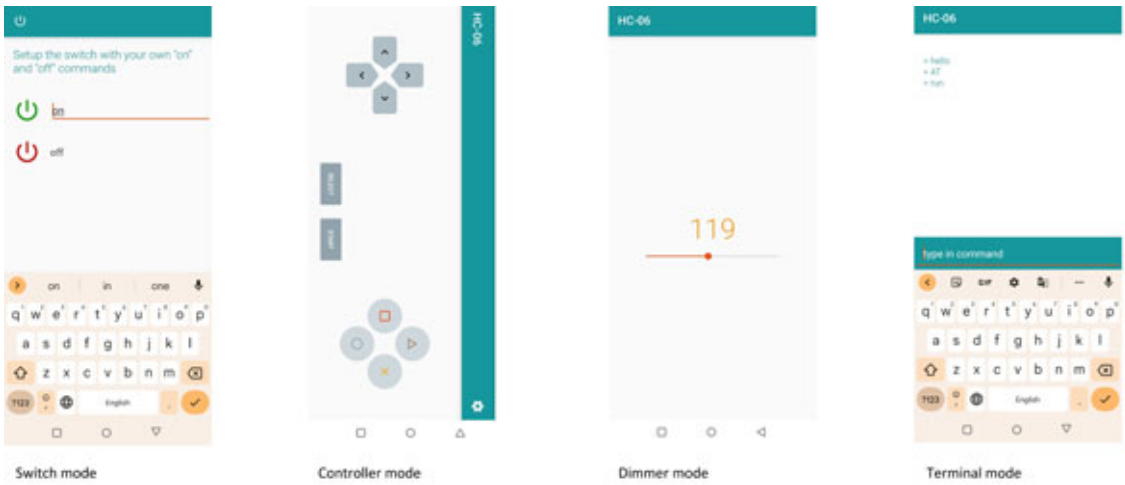


Figure 9.7: *Different modes of configuring the mobile app*

Figure 9.7 shows all the modes of configuring a Bluetooth controller app. The four different modes of configuration on the mobile app are as follows:

- **Switch mode:** Here, we get a switch on the interface. It can have two colors, and we can write any corresponding command. A very simple way to configure them is to assign “Y” and “N” to these switch colors. Whatever text is assigned to these colors, the same will be transmitted to the Arduino.
- **Controller mode:** We get a joystick type of interface in this mode, and we can send 8 different inputs to the Arduino board. This mode is very popular for designing Bluetooth controlled vehicles. We will design one such model in future recipes.
- **Dimmer mode:** This mode is specifically designed to provide analog inputs. We can send an integer from 0 to 255 using a slider.
- **Terminal mode:** This mode sends and receives the commands to the Arduino board.

Now, let us focus on the next part of the recipe: the hardware connections. We aim to control an LED using the switch mode of the app. That means if we change the color of the switch to green, the Arduino board will turn on LED, and it will be turned off on red. We should connect the 4 pins of HC-06 to the following pins of Arduino boards. The connections are also shown in *Figure 9.9*:

- VCC – to 5V power supply

- GND – to ground pin of the board
- RXD-to the TXD pin of the Arduino (digital pin 1)
- TXD – to the RXD pin of the board (digital pin 0)
- In addition to this, an LED is connected from digital pin 13 to the ground

We must ensure that the RXD and TXD pins of the Arduino are not connected while uploading the code to the board. Refer to *Figure 9.8*:

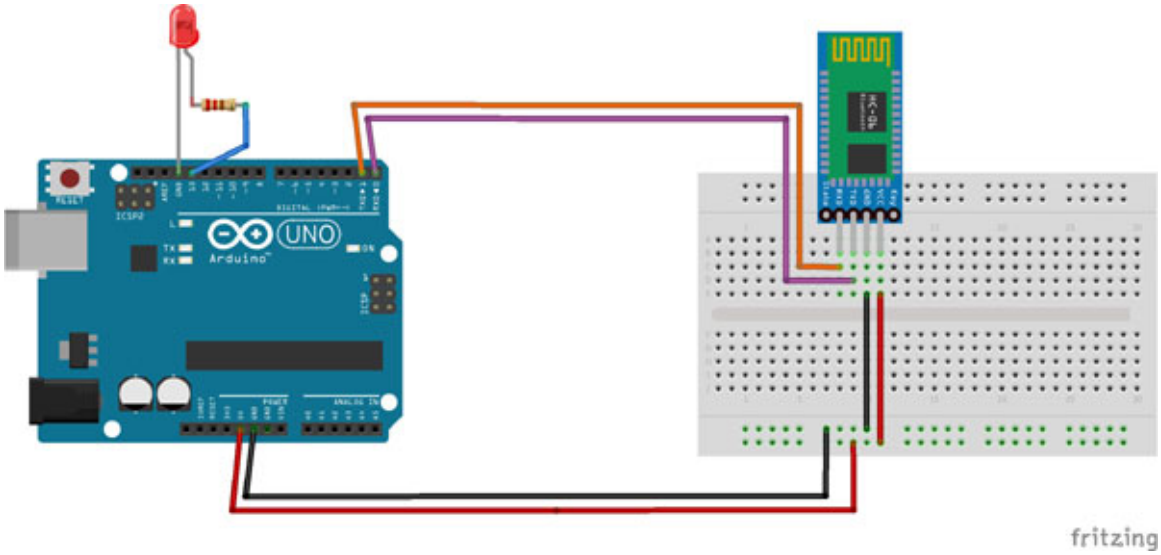


Figure 9.8: *Connecting an LED and HC-06 to the Arduino board to be controlled by a mobile app*

Now, let us focus on the third aspect of the recipe, which is the app configuration and Arduino sketch. Follow the given steps:

1. Turn on the HC-06 module by connecting only the VCC and GND pins.
2. Pair it with the mobile phone and open the Arduino Bluetooth Controller app.
3. Go to the switch mode.
4. Go to the top-right corner and configure the switches.
5. For green color, select '`Y`'.
6. For the red color, select '`N`'.
7. Write the following sketch on Arduino IDE and upload it to the Arduino board, ensuring that the RXD and TXD pins are not connected.
8. After uploading the code, connect the RXD and TXD pins and control the LED with the app.

9. The LED will turn on at the green color and off at the red color switch.

The code for the recipe is as follows:

```
char BLE_Data;
void setup()
{
 pinMode(13, OUTPUT);
 Serial.begin(9600);
}
void loop()
{
 if(Serial.available())
 {
  BLE_Data = Serial.read();
 }
 if(BLE_Data == ‘Y’)
 {
  digitalWrite(13,HIGH);
  Serial.println(“On signal received “);
  delay(1000);
 }
 else if (BLE_Data == ‘N’)
 {
  digitalWrite(13,LOW);
  Serial.println(“OFF signal received “);
  delay(1000);
 }
 else
 {
  digitalWrite(13, LOW);
  Serial.println(“Not a valid signal received”);
  delay(1000);
 }
}
```

We can observe that in the preceding sketch, we have also used Serial communication for every situation. Therefore, you can open the serial monitor

and check the output on it. It helps us debug whether we receive the app's correct codes.

Recipe 4: Visualize serial monitor information over Bluetooth

The previous recipe is a building block for any Bluetooth controlled system using an Arduino board. We understood three major parts of designing any such system. Now, let us explore a few more features of the app and design recipes based on these. In this recipe, we will use the terminal mode of the app. Using this mode, we will send the data from the app to the board, control the LED connected to the board and send a message back to the app. Therefore, we can visualize the flow of information from both devices.

This mode can be tested very easily, without making major changes to the previous recipe. We will be using the same setup for hardware. Therefore, we will connect the circuit as shown in *Figure 9.8*. Now, let us focus on the app configuration and Arduino sketch:

- Turn on the HC-06 module by connecting only the VCC and GND pins.
- Pair it with the mobile phone and open the Arduino Bluetooth Controller app.
- Write the following sketch on Arduino IDE and upload it to the Arduino board, ensuring that the RXD and TXD pins are not connected.
- After uploading the code, connect the RXD and TXD pins and control the LED with the app.
- The code will start executing.
- Go to the terminal mode.
- The terminal will start showing the output.
- Use the keyboard to send commands from the app.
- We will observe the relevant outputs on the same.

The sketch for the recipe is as follows:

```
char BLE_Data;
char lastData;
void setup()
{
```

```
  pinMode(13, OUTPUT);
  Serial.begin(9600);
}
void loop()
{
  if(Serial.available())
  {
   BLE_Data = Serial.read();
  }
  if(BLE_Data == ‘H’)
  {
   digitalWrite(13,HIGH);
   if(lastData!= ‘H’)
    Serial.println(F(“The LED is turned on now”));
   lastData = BLE_Data;
   delay(1000);
  }
  else if (BLE_Data == ‘L’)
  {
   digitalWrite(13,LOW);
   if(lastData!= ‘L’)
    Serial.println(F(“The LED is turned off now”));
   lastData = BLE_Data;
   delay(1000);
  }
  else
  {
   digitalWrite(13, LOW);
   Serial.println(“Not a valid signal received”);
   delay(1000);
  }
}
```

The sketch checks the message sent from the app, and if the signal received is ‘H’, then the LED is turned on, and the acknowledgement is sent to the app in terminal mode. A similar pattern is observed for the ‘L’ signal.

Figure 9.9 shows the information being displayed on the app for various commands sent through the keyboard:

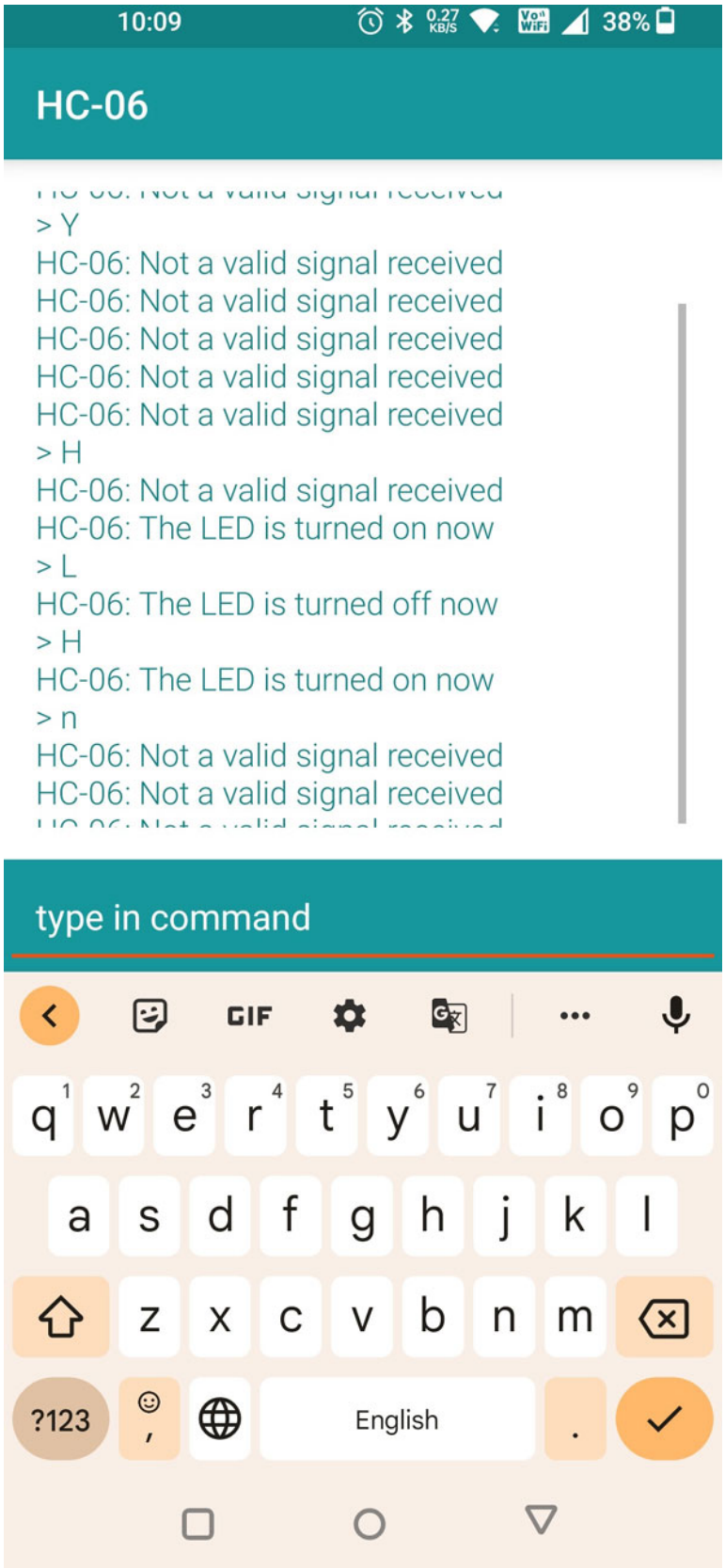

10:09
38%
HC-06
> Y
HC-06: Not a valid signal received
HC-06: Not a valid signal received
HC-06: Not a valid signal received
HC-06: Not a valid signal received
HC-06: Not a valid signal received
> H
HC-06: Not a valid signal received
HC-06: The LED is turned on now
> L
HC-06: The LED is turned off now
> H
HC-06: The LED is turned on now
> n
HC-06: Not a valid signal received
HC-06: Not a valid signal received
type in command
GIF
English
?123

Figure 9.9: *Different outputs printed on the terminal of the Arduino Bluetooth Controller app*

Recipe 5: Smartphone controlled car

We are building our competencies with Bluetooth-based recipes slowly. In this recipe, we will design a toy car that can be controlled with our smartphones. The communication between the smartphone and the car will be through Bluetooth. We will use the concepts used in the previous recipes, with a different configuration of the app and a new set of hardware connections.

Figure 9.10 shows the hardware connections for the Bluetooth controlled car. We have connected the HC-06 to the RXD and TXD pins (digital pins 0 and 1, respectively) of the Arduino board. We need to control two DC motors. Therefore, we have used digital pins 3 and 4 as output pins. The output from these pins is fed to the L293D motor driver. The outputs of the motor driver control the motors. Refer to the following figure:

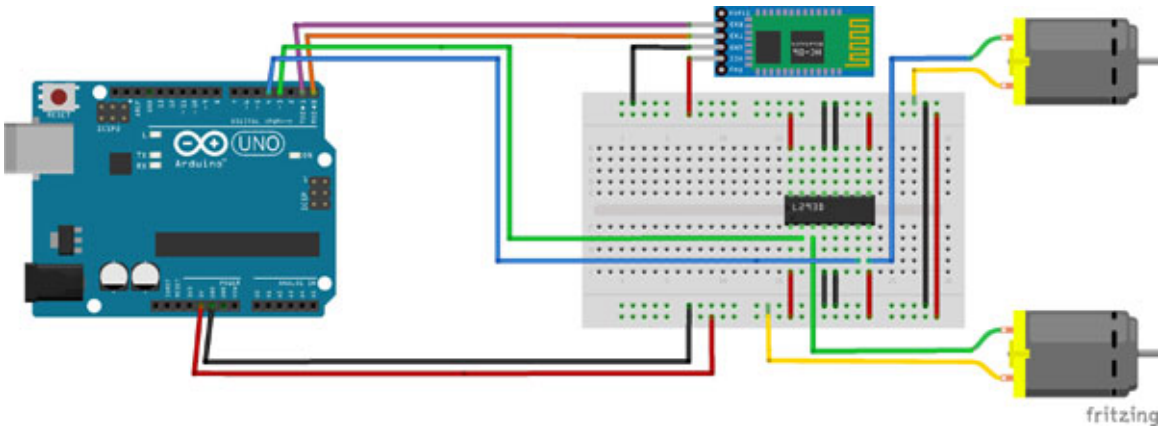


Figure 9.10: *Connections for a Bluetooth controlled car with Arduino board*

Once the hardware setup is done, let us follow these steps to configure the app and write the sketch for the recipe:

1. Turn on the HC-06 module by connecting only the VCC and GND pins.
2. Pair it with the mobile phone and open the Arduino Bluetooth Controller app.
3. Open the app and open the Controller mode.
4. Configure the left, right, up, and down buttons as L, R, F, and S, respectively.

5. Write the following sketch on Arduino IDE and upload it to the Arduino board, ensuring that the RXD and TXD pins are not connected.
6. After uploading the code, connect the RXD and TXD pins and control the LED with the app.
7. The code will start executing, and you can control the car using these 4 buttons.

The sketch for the recipe is following. In this recipe, we send four different messages from the app. The message L is for turning left, R is for turning right, and F is for moving forward. We have used S to stop the car. The code uses simple if-else statements to implement the logic:

```
char BLE_Data;
void setup() {
  pinMode(3, OUTPUT); // for the left motor
  pinMode(4, OUTPUT); // for the right motor
  Serial.begin(9600);
}
void loop() {
  if(Serial.available())
  {
   BLE_Data = Serial.read();
  }
  if(BLE_Data == ‘F’)
  {
   digitalWrite(3,HIGH);
   digitalWrite(4,HIGH);
   delay(100);
  }
  else if(BLE_Data == ‘L’) // turning left by stopping left motor.
  {
   digitalWrite(3,LOW);
   digitalWrite(4,HIGH);
   delay(100);
  }
  else if(BLE_Data == ‘R’) // turning right by stopping the right
  motor.
```

```
  {
   digitalWrite(3,HIGH);
   digitalWrite(4,LOW);
   delay(100);
  }
  else if(BLE_Data == 'S') // turning right by stopping the right
  motor.
  {
   digitalWrite(3,LOW);
   digitalWrite(4,LOW);
   delay(100);
  }
  else
  {
   // in case of an invalid command, stop the car.
   digitalWrite(3,LOW);
   digitalWrite(4,LOW);
   delay(100);
  }
 }
```

Recipe 6: Bluetooth controlled servo motor

We have designed quite a few interesting recipes using the Bluetooth module and Arduino combination. We have observed that within the range of a few meters, we can control the devices using wireless control through a mobile app. In this recipe, we will explore one more feature of the app and control the angle of rotation of a servo motor using the mobile app.

A servo motor can rotate from 0 to 180 degrees. We can use the app's dimmer mode to control the rotation angle.

Figure 9.11 shows the hardware connections of an HC-06 with an Arduino board and a servo motor. We have connected the servo motor to digital pin 3 in this recipe.

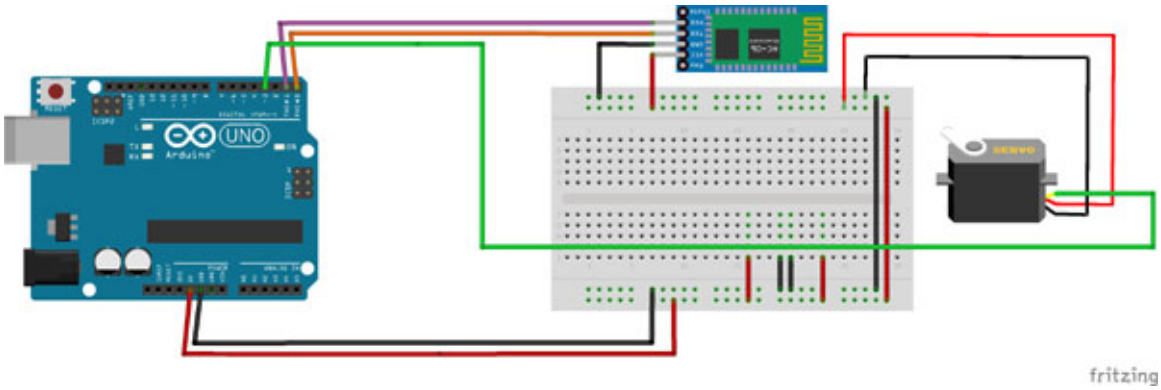


Figure 9.11: *Controlling a servo motor angle with a mobile app through Bluetooth communication*

In the dimmer mode, we can slide the bar and give analog inputs to the board. Here, we can slide and generate numbers from 0to 255. Some apps have this slider from 0 to 100. We read the number coming from the Bluetooth module in **`BLE_Data`**. We have attached the servo moto at digital pin. The sketch for the recipe is given as follows; we simply read the value coming from the Bluetooth module and write that as the angle of rotation for the servo motor in the code:

```
#include <Servo.h>
Servo servo_3;
int BLE_Data;
void setup()
{
  servo_3.attach(3, 500, 2500);
  Serial.begin(9600);
}
void loop()
{
  if(Serial.available())
  {
    BLE_Data = Serial.read();
    Serial.println(BLE_Data);
    servo_3.write(BLE_Data);
    delay(10);
  }
}
```

Conclusion

In this chapter, we learned about one of the most popular wireless communication protocols: Bluetooth. We discussed different modules that can be connected to an Arduino board to enable Bluetooth communication. We discussed the features of different modules and their connections with the Arduino board and a PC. To enable wireless communication, we use a smartphone. We learnt how to install an app to control the Bluetooth modules connected to the Arduino board and designed a few interesting recipes.

In the next chapter, we will continue learning about wireless communication techniques and understand how to transmit data using Wi-Fi with an Arduino board.

Points to remember

- Ensure that the Bluetooth module is connected to only VCC and GND while detecting on a PC or mobile phone.
- RXD and TXD should be connected appropriately for AT command mode.
- RXD and TXD should be flipped to work in data transfer mode.
- Ensure that RXD and TXD are not connected while uploading the code.
- Motors drivers should have a separate power supply.
- The ground of all the power supplies in the system must be connected to one common point.

Multiple choice questions

1. **How many pins are present on an HM-10 Bluetooth module?**

    a. 4

    b. 6

    c. 5

    d. 8

2. **Which pins are default serial communication pins on an Arduino board?**

a. 0, 1
b. A0, A1
c. 12,13
d. 8,9

3. **For AT mode, the RXD pin of the Bluetooth module should be connected to ____ on Arduino.**

   a. TXD
   b. GND
   c. VCC
   d. RXD

4. **For data transfer via Bluetooth, the RXD pin of the Bluetooth module should be connected to the _____ pin on the Arduino board.**

   a. TXD
   b. GND
   c. VCC
   d. RXD

5. **Bluetooth communication is _____ communication.**

   a. Parallel
   b. Ultrafast
   c. serial
   d. long distance

Answers

1. **b**
2. **a**
3. **d**
4. **a**
5. **c**

CHAPTER 10

Wi-Fi Connection Using Arduino

Introduction

In this chapter, we will discuss one of the most widely used wireless communication protocols in today's world: Wi-Fi. As we know, Arduino UNO doesn't have support for Wi-Fi communication, so first, we will identify the different options for establishing Wi-Fi communication with microcontrollers. We will learn about a new series of boards. We will learn about ESP32 and ESP8266. Then, we will establish the communication, connect the device to the Wi-Fi and send the data to a web portal.

Structure

This chapter will have the following recipes related to the Wi-Fi connectivity:

- Different options to connect to Wi-Fi via microcontrollers
- Setting Arduino IDE for ESP32 or ESP8266 boardsLED blink using ESP32Setting Wi-Fi using ESP32A simple web server design using ESP32
- Logging real-time temperature and humidity values on the web
- Internet clock design using ESP32

Objectives

By the end of this chapter, you will be able to do the following:

- Know about different microcontrollers with inbuilt Wi-Fi support
- Know the difference between ESP32 and ESP8266 boards
- Connect ESP8266 with the Wi-Fi at their home or workspace
- Transmit data via Wi-Fi to an internet server
- Connect a DHT22 sensor with ESP8266

Recipe 1: Different options to connect to Wi-Fi via microcontrollers

In this chapter, we will focus on Wi-Fi communication through microcontrollers. Arduino is a very popular microcontroller, but it does not support Wi-Fi communication. We can add the Wi-Fi shield to the Arduino and make it connect to any Wi-Fi network. These shields are not only expensive but also less reliable for robust communication. Therefore, we can look at the other microcontrollers that can be programmed using Arduino IDE and establish reliable Wi-Fi communication. So, first, let us explore our options for setting up Wi-Fi communication through the microcontrollers.

ESP32 and ESP8266 are the most popular choices for Wi-Fi communication through microcontrollers. *Figure 10.1* shows the ESP32 and ESP 8266 boards. If we look at the core of these boards, they are tiny and can be attached to any other component. We can use these in coordination with an Arduino board as well. These boards come with onboard Wi-Fi support. The development boards of ESP32 and ESP8266 are also named Node MCU. The name "*NodeMCU*" combines "*node*" and "*MCU*" (*Micro-Controller Unit*). Therefore, you will also read Node MCU terminology at various technical platforms related to ESP32 and ESP8266 development boards. Refer to the following figure:

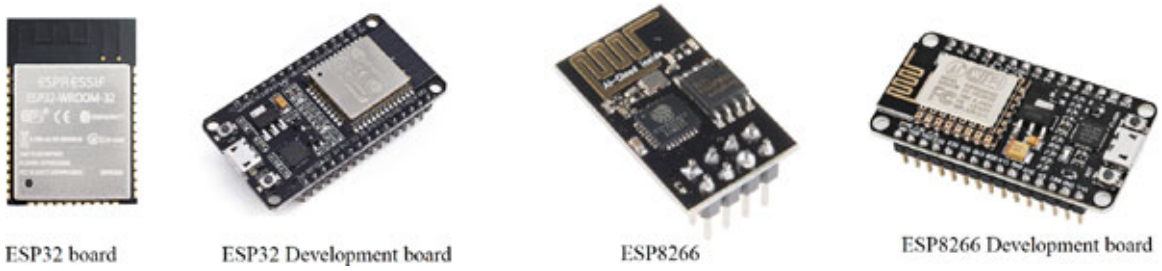


Figure 10.1: *ESP32 and ESP8266 boards shown in their core form and as development boards*

Let us compare these boards in terms of their technical specifications so that we can choose one of them for our respective applications.

Table 10.1 shows the technical comparison between the ESP32 and ESP8266:

Feature	ESP8266	ESP32
Wi-Fi version	HT20	HT40
Bluetooth	NA	Bluetooth 4.2 and BLE
Typical Frequency	80MHz	160MHz

SRAM	Not available	Available
FLASH	Not available	Available
Number of General-purpose input output pins	17	34
PWM pins	8	16
SPI/I2C/I2S/UART	2/1/2/2	4/2/2/2
ADC	10 bit	12 bit
controller area network (CAN)	Not available	Available
Touch sensor	Not available	Available
Temperature Sensor	Not available	Available
Ethernet MAC Interface	Not available	Available

Table 10.1: *Technical comparison of ESP32 and ESP8266*

By observing the features from *Table 10.1*, we can easily say that ESP32 is a superior board compared to ESP8266. However, if we aim to establish Wi-Fi communication and design simple systems, then both are equally capable. The user can take any of these boards depending on the computational complexities and the number of input-output pins required for the project. Both boards are available at an almost equal price point.

Setting Arduino IDE for ESP32 or ESP8266 boards

We have decided that we are going to use ESP32 or EP8266 for establishing Wi-Fi communication through the microcontroller. In this recipe, we will focus on the configuration of Arduino IDE for connecting and programming one of these boards. The process must be completed only once, and then the IDE will work smoothly with the boards in the future. The following are the steps for ESP8266, and a similar sequence of steps can also help you configure the IDE for ESP32:

1. Connect the ESP8266 to your computer.
2. Open the device manager and check the ports. Note down the port number for the port with the name **Silicon Labs CP210x USB to UART Bridge.** *Figure 10.2* shows the device manager and indicates that the ESP8266 is detected at COM5 with the above-mentioned name.
3. Open Arduino IDE; refer to the following figure:

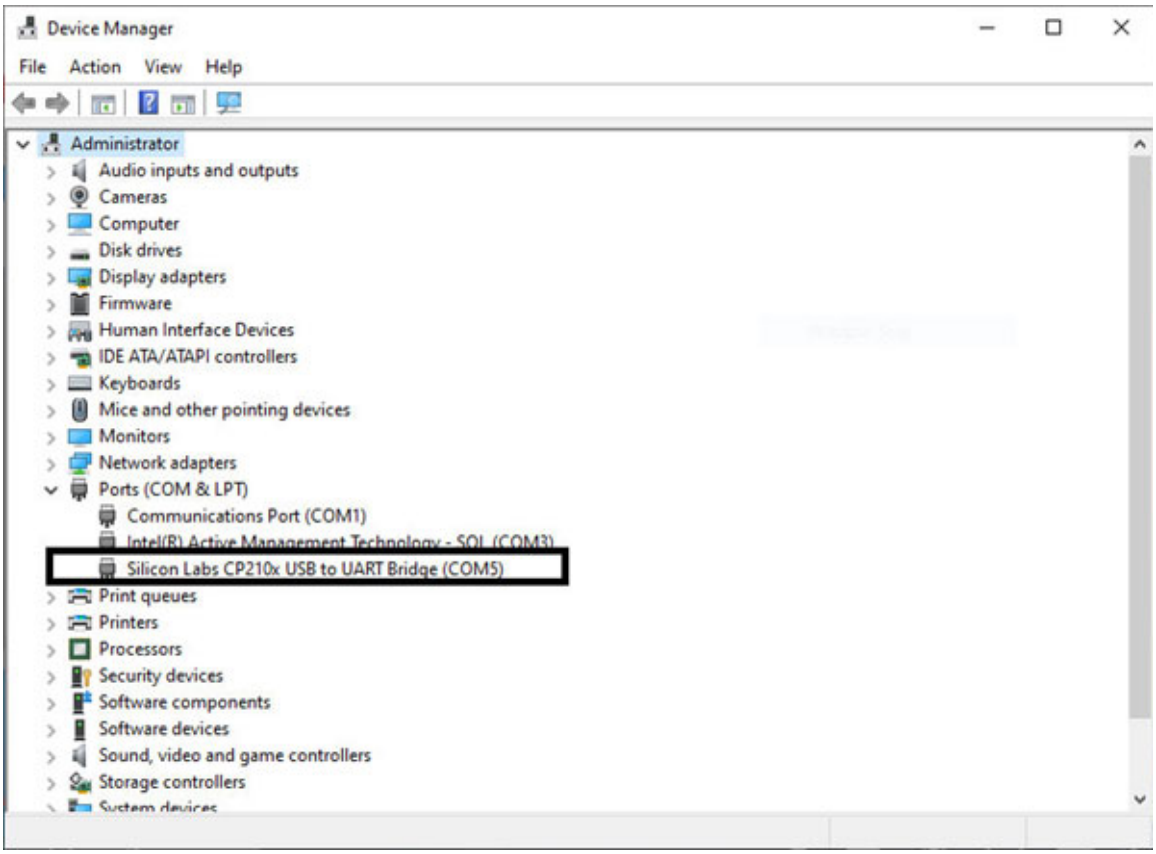


Figure 10.2: *The device manager indicated the detection of ESP8266 on the computer*

4. Open the URL **https://github.com/esp8266/Arduino** and scroll down to the "**`Installing with board Manager`**" Section. This URL provides us with the settings for adding a new hardware board to the Arduino IDE.
5. Start Arduino and open the **`Preferences`** window.
6. Enter **https://arduino.esp8266.com/stable/package_esp8266com_index.json** into the **`File`**>**`Preferences`**>**`Additional Boards Manager URLs`** field of the Arduino IDE. *Figure 10.3* shows the preferences tab filled with the given URL in Arduino IDE:

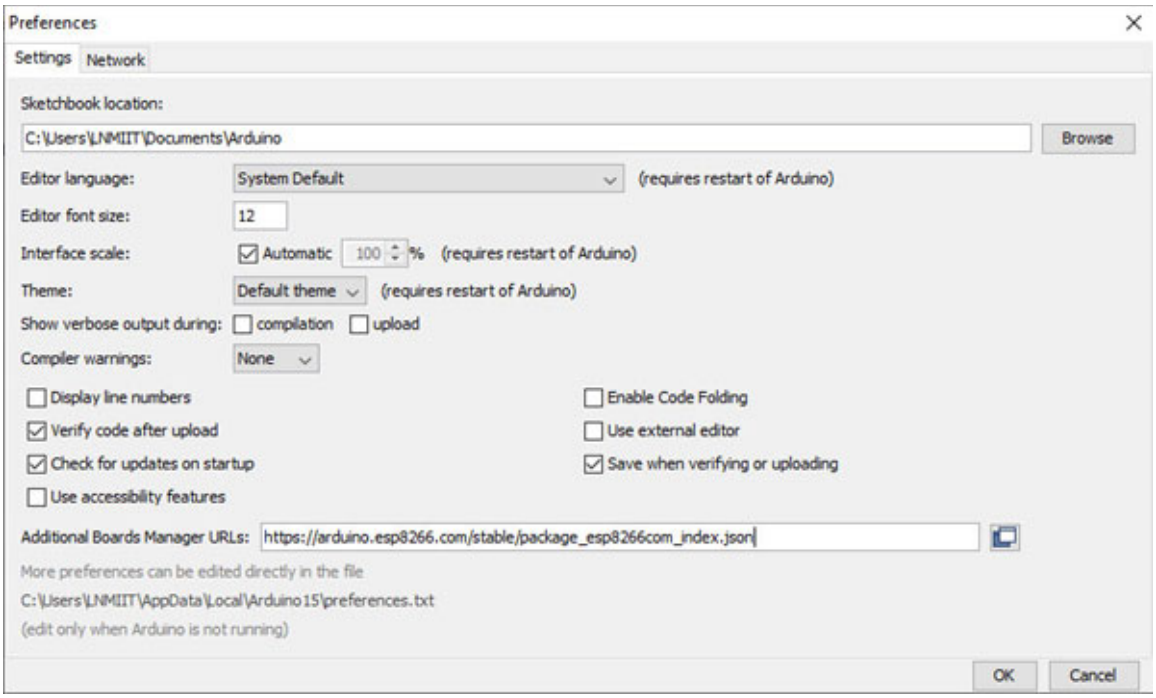


Figure 10.3: *Preferences tab of the Arduino IDE and adding more URLs for additional boards*

7. Open Boards Manager from **`Tools`** > **`Board menu`** and install the ESP8266 platform. Search ESP8266 and click on the **`install`** button to install the boards in the family. *Figure 10.4*. shows the installation process in progress for ESP8266:

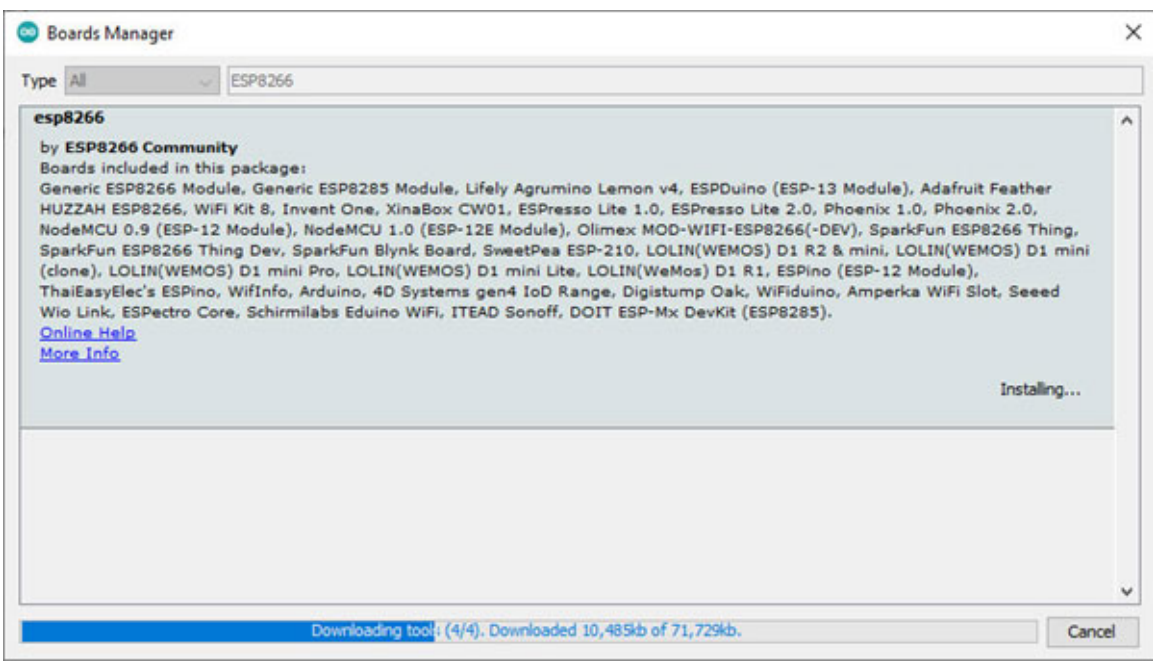


Figure 10.4: *Installing a new board family through the board manager*

8. Go to **`Tools`** > **`Board menu`** and then go to ESP Boards. Scroll down and find NodeMCU1.0. *Figure 10.5* shows the path for the same:

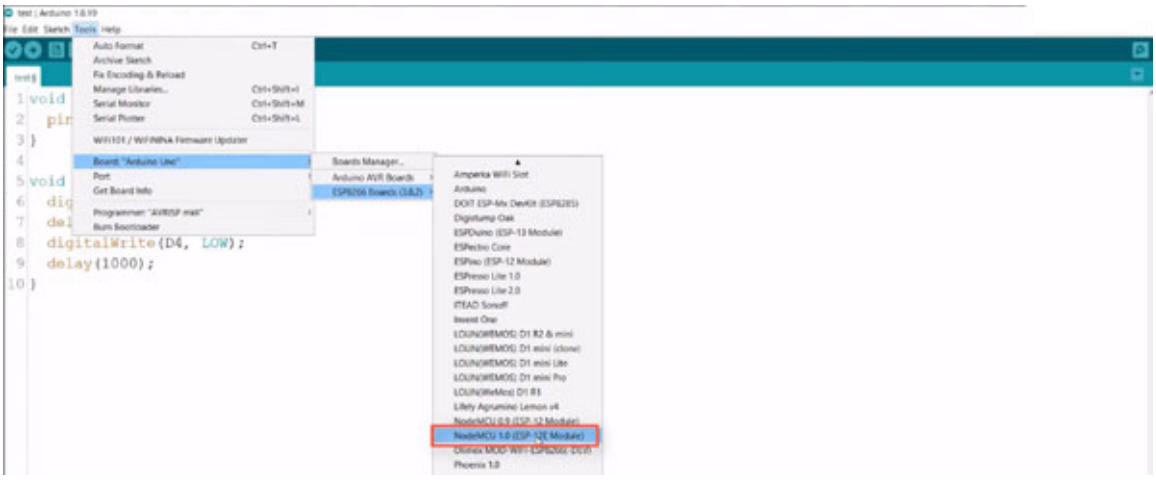


Figure 10.5: *Selecting the correct NodeMCU board for the ESP8266 development board*

9. Go to **`Tools`** > **`Port`** and select the port on which ESP8266 was detected. In this example, it was COM5.

With these steps, the IDE is configured to connect with the ESP8266 development board. Now, we can write, compile, and upload the codes from the Arduino IDE.

Recipe 2: LED blink using ESP8266

LED blink is typically the first code to test any programmable hardware board. We can test this code in two ways. In the first method, we'll blink an onboard LED; in the second, we will connect an external LED and then blink it. In the previous recipe, we observed that ESP8266 is a superior board compared to ESP32. Therefore, we will blink an LED using the ESP8266 development board in this recipe.

Figure 10.1 shows the ESP8266 development board, with 30 physical pins connected to the board. There are different types, numbers and names for these pins, which are associated with the functionality, coding, and connections. Therefore, it is vital to know these pins' locations, types, and the respective names before testing anything on the hardware.

Figure 10.6 shows the pinout of the ESP8266 board. The 30 hardware pins can function in 10 different categories. Two pins are kept reserved as well. We can observe that there are 16 General Purpose Input/Output (GPIO) pins, a few

pins for every type of communication protocol, and all these pins have different names printed on the hardware board. Refer to the following figure:

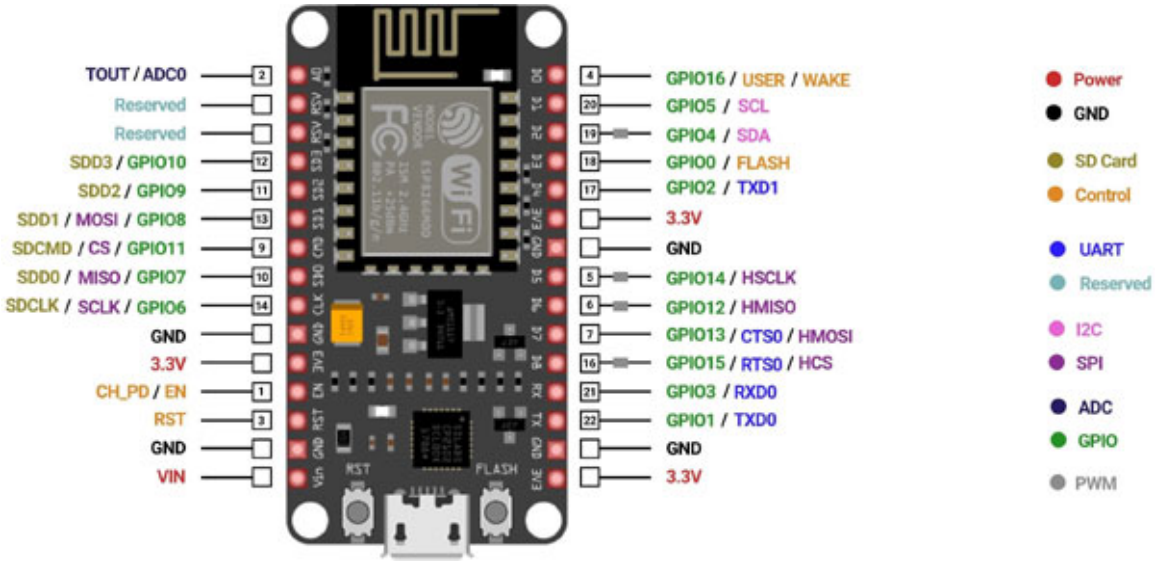


Figure 10.6: *The pinout of NodeMCU. Different types of pins on the board*

The differences between printed and pinout names can confuse a user while programming the board. Therefore, it is important to know the relations between these names.

Table 10.2 shows the relations between the name of the pin printed on the board, its function, the number to be used in Arduino IDE, and the alias name that can be used in the IDE. For example, the pin named D3 on the board is a GPIO pin. This pin can be used for flashing the board as well. This pin will be programmed using the aliasD3 on the Arduino IDE. In the same table, we can observe that D0 is the pin associated with the LED_BUILTIN. Therefore, if we program our board at pin D0, we can blink the onboard LED. On the ESP8266 board, the BUILTIN LED is located near the power supply, opposite to RST pin.

Pin Name on the Board	Function	Pin Number in Arduino IDE	Alias Name in Arduino IDE
D3	GPIO 0	0	D3
TX	GPIO 1	1	D10
D4	GPIO 2	2	D4
RX	GPIO 3	3	D9

D2	GPIO 4	4	D2
D1	GPIO 5	5	D1
D6	GPIO 12	12	D6
D7	GPIO 13	13	D7
D5	GPIO 14	14	D5
D8	GPIO 15	15	D8
D0	GPIO 16	16	D0, LED_BUILTIN
A0	ADC0	A0	analog_ip

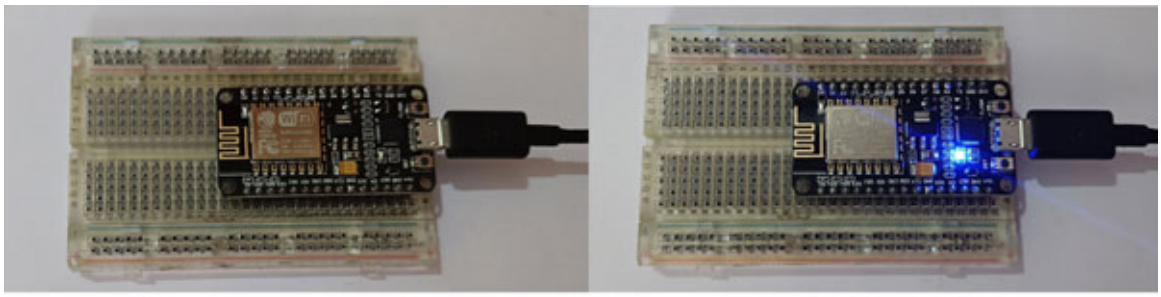
Table 10.2: *Different pins on NodeMCU and their names in the Arduino IDE*

Once we have a clear idea about the pin locations and their names in the Arduino IDE, the LED blink program is very easy to write, compile, and upload. The code is exactly the same as that for an Arduino LED blink code. The only difference is the name of the pin to be used. The code for the same is mentioned as follows:

```
void setup() {
  pinMode(D0, OUTPUT);
}
void loop() {
  digitalWrite(D0, HIGH);
  delay(1000);
  digitalWrite(D0, LOW);
  delay(1000);
}
```

Open IDE, write the code, compile it, and upload it by ensuring that we have selected the correct board and the port. Compiling and uploading the ESP8266 codes take more time compared to Arduino. While the code is uploaded, an LED near pin D0 blinks with lower intensity, indicating that the code transfer is taking place. While on successful uploading, the onboard LED starts blinking.

Figure 10.7 shows two cases. In the first one, the code is uploaded on the board, while in the second, the BUILTIN LED blinks:



Uploading the code on the ESP8266 development board. Blinking the on-board LED.

Figure 10.7: *Uploading a code on ESP8266 and blinking the onboard LED*

Recipe 3: Connecting an ESP8266 to Wi-Fi

In his recipe, we will utilize the main function of ESP8266: Wi-Fi communication. We will learn the tools and techniques required to establish communication between our board and a network. We will use the Wi-Fi library to utilize various functions related to the communication protocol.

We do not need any hardware connections to set up the Wi-Fi communication through the ESP8266 board. So, we simply connect the board to the PC and upload the sketch. We will use the following libraries to enable Wi-Fi communication:

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
```

These are already included in the IDE with the ESP8266 board settings, so we need not install them separately. This recipe aims to connect our ESP8266 to a Wi-Fi network and print the stats on the serial monitor. A Wi-Fi network is declared by its service set identifier (SSID) and the password key. In this example, we have set up a testing network through the hotspot of a mobile phone. The SSID is "**`OnePlusNord2`**", and the password is "**`test1234`**". To understand the exhaustive features of the Wi-Fi library, you can visit **https://www.arduino.cc/reference/en/libraries/wifi/.** This documentation shows all the functions related to the library. We will use only a few in the setup process.

The code for the recipe is as follows:

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
```

```
const char* ssid = “OnePlusNord2”; // Replace it with your own
SSID
const char* password = “test1234”; // replace it with your
password
int status;
WiFiClient client;
void setup() {
 Serial.begin(9600);
 delay(10);
 status = WiFi.begin(ssid, password);
}
void loop() {
 Serial.println(status);
 int x = WiFi.status();
 Serial.println(x);
 delay(1000);
 if ( x == 3) {
  Serial.println(“Connected to the Wi-Fi now”);
  Serial.print(“Wi-Fi SSID is = “);
  Serial.println(ssid);
  delay(1000);
 }
 else{
  Serial.println(“Not connected to the network”);
  delay(1000);
 }
}
```

In this code, we have defined two variables: SSID and password. The **`WiFi.begin()`** function starts communicating with the given SSID and password. Additionally, we stored the status in a variable named **`status`**. In the **`loop()`** function, we used another function called **`WiFi.status()`**, to check the status of the connectivity. The function returned 3 when connected and 7 when not connected. These values can easily be tested using the **`Serial.print()`** function. Then, we used the **`if-else`** statement to print the status of the connectivity.

Figure 10.8 shows the logs of the serial monitor at different stages. Initially, the network is not switched on, so the message "**`Not connected to the network`**" is printed. Once we enable the hotspot with the same SSID and password, the ESP8266 is connected to it, and the message is printed to show the connectivity status. The figure also includes the screenshot from the mobile, showing that one device is connected to the given network now.

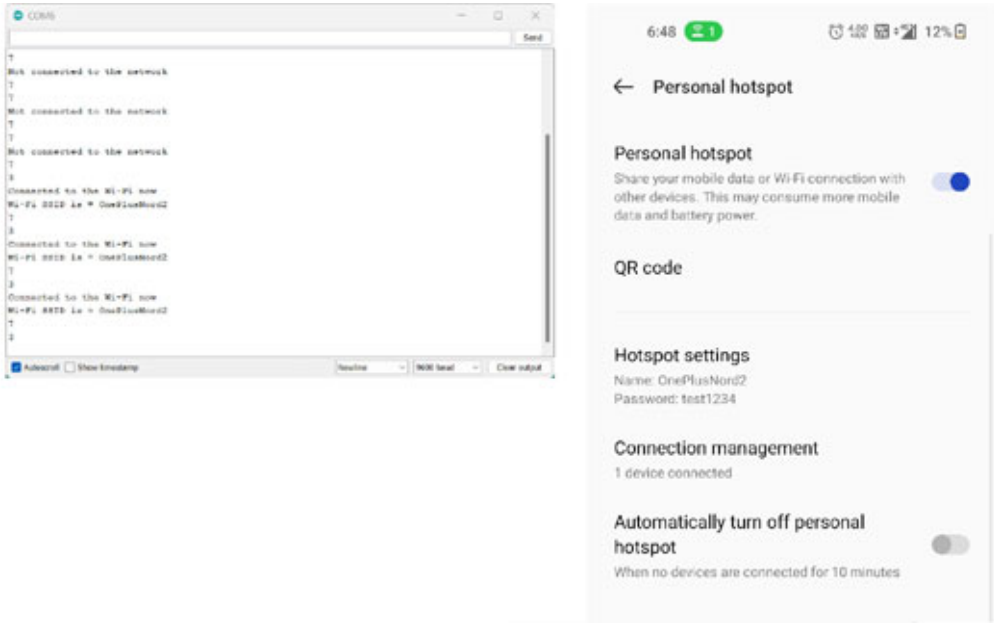


Figure 10.8: *Serial monitor and the mobile phone showing the connectivity status of the ESP8266*

Recipe 4: Logging real-time light intensity values on the web

We successfully established Wi-Fi communication with our ESP8266 development board in the previous recipe. In this recipe, we will develop the first application to enter the **Internet-of-Things** (**IoT**) field. The field of IoT is the latest emerging area in the modern world, and it is the backbone for all the smart systems being developed around us. It helps us design smart homes and automated factories and monitor many parameters in real time while sitting anywhere in the world. We connect our devices and appliances (called things) to the internet (via Wi-Fi) to establish the IoT network. The devices transmit and receive commands from the user to control operation.

In this recipe, we will transmit the light intensity values recorded through a light sensor to a web portal and plot it in real time over a period. We can

explore different options to send our sensor data to the internet. We can create our own web server and upload our data there, or we can use an existing cloud-based server to upload our data. One of the most popular data logging servers is available at **https://thingspeak.com/**. ThingSpeak is a platform provided by MathWorks, and it is available as a free service for small non-commercial projects (<3 million messages/year or ~8,200 messages/day). Four annual license types are offered for larger projects or commercial applications: Standard, Academic, Student, and Home. Therefore, it is perfect for beginners to test their IoT projects.

Creating an account on ThingSpeak is free. Follow these steps to create an account, and get ready to transfer data:

1. Go to **https://thingspeak.com/** and click in the top-right corner to sign in. *Figure 10.9* shows the path to create a new account if you do not have one. Else, sign in to your existing account.

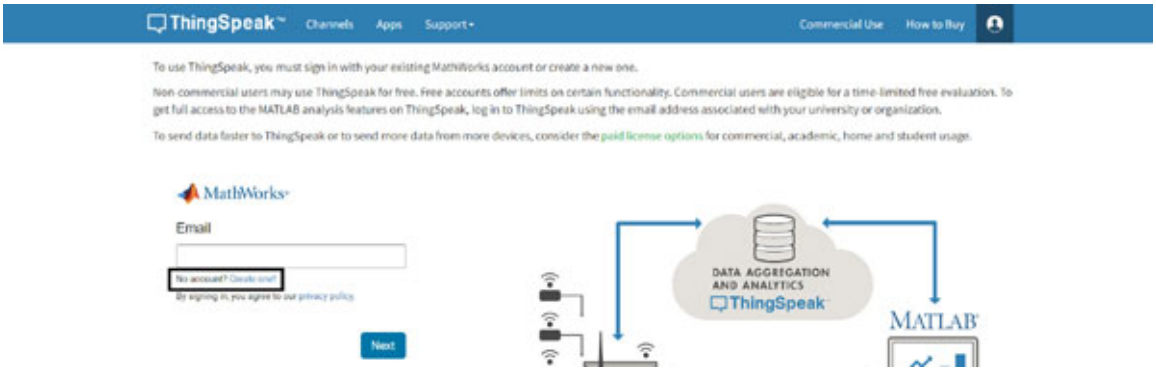


Figure 10.9: *Creating an account on ThingSpeak*

2. ThingSpeak records data in the form of channels. You can access a channel to record one type of data for any period of time. After successful login, your previous channels will be visible on the dashboard, and there will be an option to create a new channel. *Figure 10.10* shows the option to create a new channel.
3. Click on the new channel button to create a new channel.

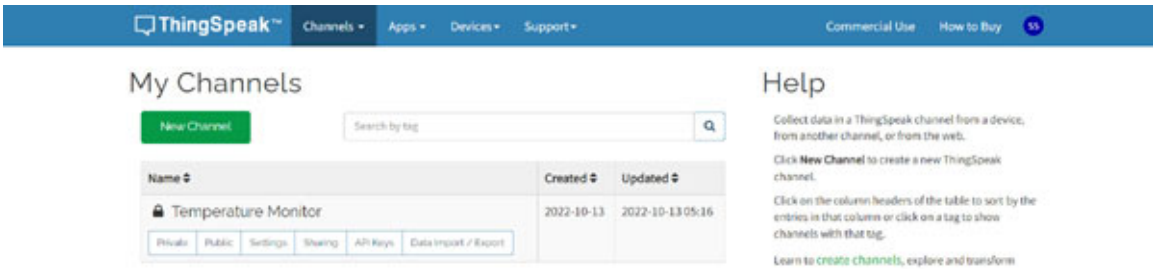


Figure 10.10: *Creating a new channel on ThingSpeak*

4. When the next page opens, give a name for your channel that is relevant to the type of data to be uploaded. You can upload more than one type of data in different fields of a channel. Here, we are recording only one field. *Figure 10.11* shows the creation of a new channel called **`Light Intensity`**. Scroll till the end and click on **`save the channel`**.

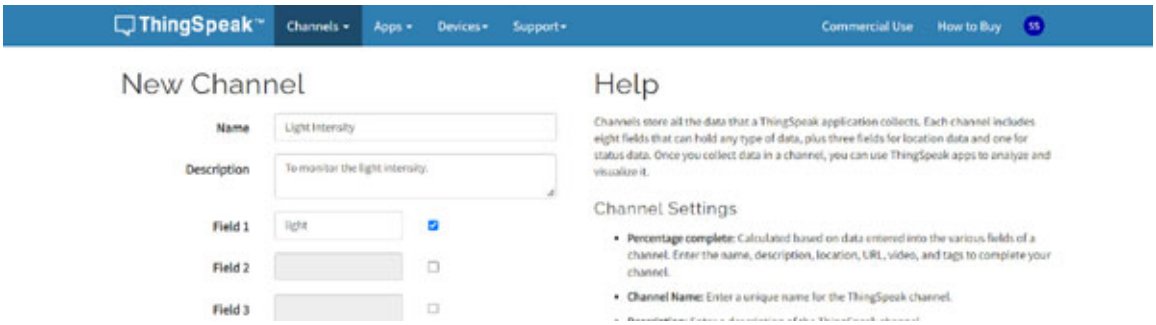


Figure 10.11: *Creating a new channel on ThingSpeak*

5. Go to the next page and observe that a unique number is allocated to your channel.
6. Go to the API keys tab, and you will also have a unique API key. We have to note down both of these values. *Figure 10.12* shows both of these values on the ThingSpeak portal:

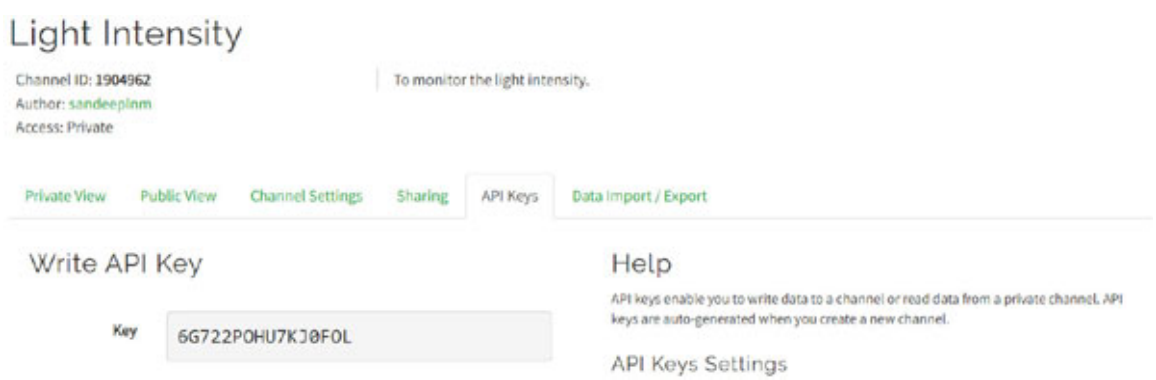


Figure 10.12: *Unique channel and API keys generated for data transfer*

7. **Connect a light sensor** (**LDR**) in series with a 10kΩ resistor and provide the voltage divider output at pin A0. The circuit connections are shown in *Figure 10.13*:

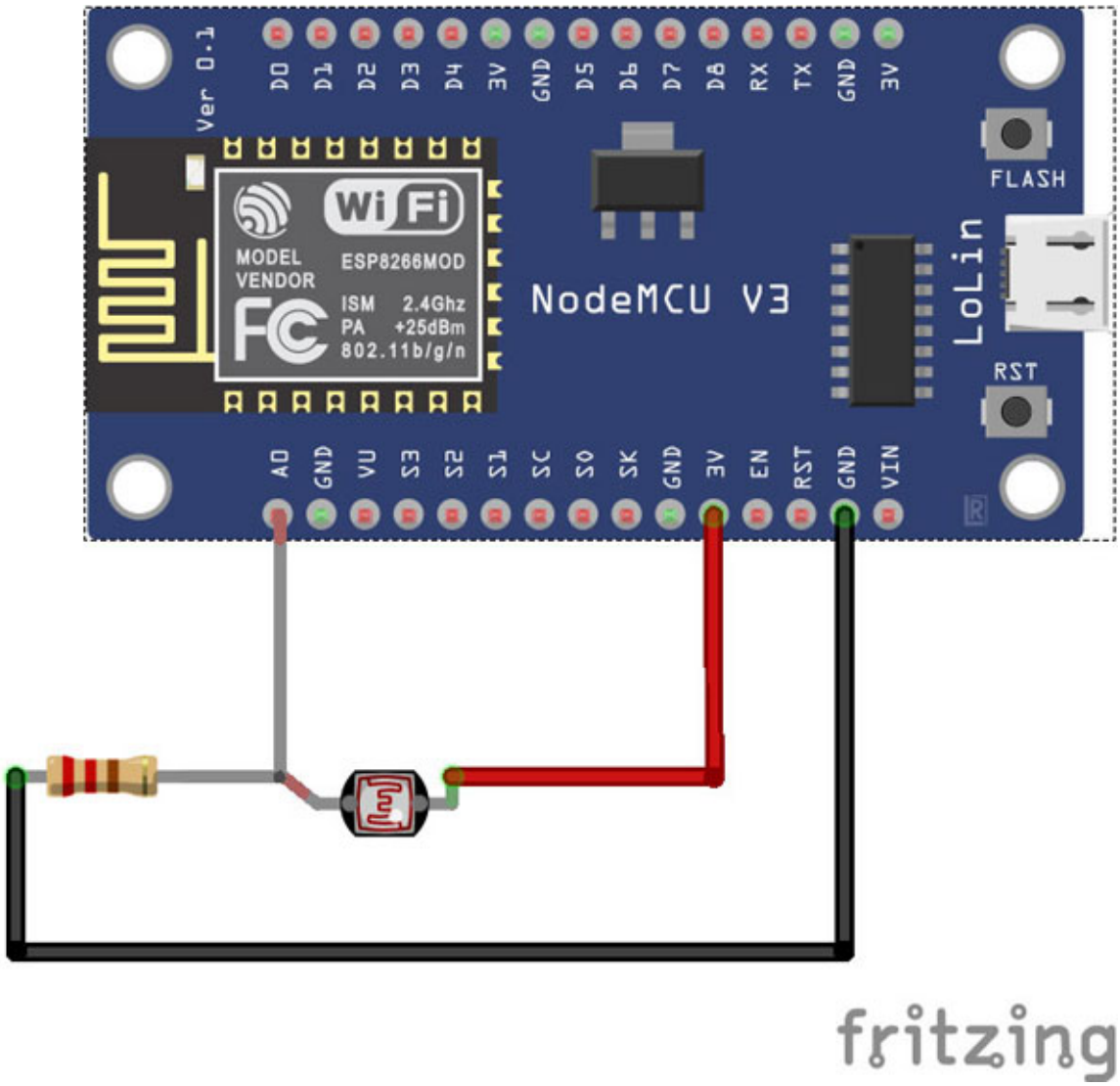


Figure 10.13: *Circuit connections to connect a light sensor to the analog pin of ESP8266*

8. Open Arduino IDE and install the **`ThingSpeak`** library from the library manager. *Figure 10.14* shows the installation process to install the **`ThingSpeak`** library:

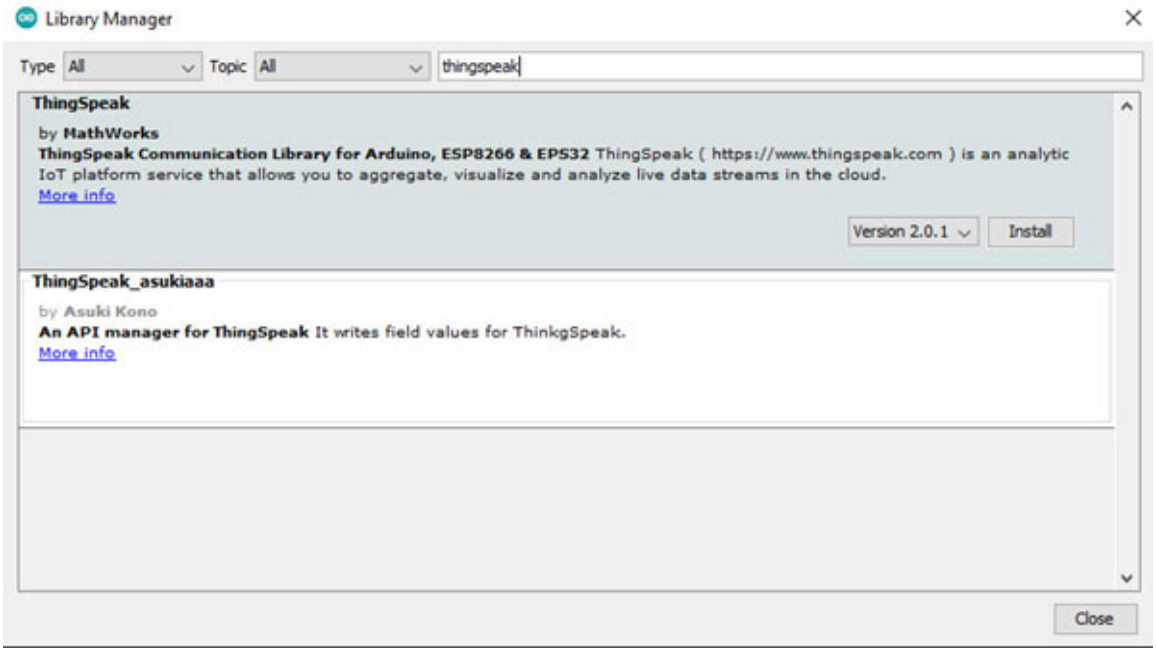


Figure 10.14: *Installing the ThingSpeak library on the Arduino IDE*

9. Write the code as given after the last step and run it on ESP8266.
10. Your data will be recorded on the **`ThingSpeak`** server.
11. Visualization will be available in real time on the portal.
12. We can go to the Private View tab and Select Export Recent Data to save this data in CSV format.

The code for the recipe is as follows:

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ThingSpeak.h>
const char* ssid = “OnePlusNord2”;
const char* password = “test1234”;
int light;
WiFiClient client;
unsigned long myChannelNumber = 1904962;
const char * myWriteAPIKey = “6G722POHU7KJ0FOL”;
void setup() {
  Serial.begin(9600);
  pinMode(A0, INPUT);
  delay(10);
```

```
  WiFi.begin(ssid, password);
  ThingSpeak.begin(client);
}
void loop() {
  light = analogRead(A0);
  Serial.print("Light intensity value = ");
  Serial.println(light);
  ThingSpeak.writeField(myChannelNumber, 1, light, myWriteAPIKey);
  delay(100);
}
```

In this code, we are starting the Wi-Fi in the beginning. We mention the unique channel name and API key to connect the ThingSpeak channel with our code. We read the analog value of light intensity and store it in the variable **`light`**. Then we start transmitting the values to the created channel. The same is printed on the serial monitor as well.

Figure 10.15 shows the real-time data logged on the ThingSpeak server and serial monitor simultaneously:

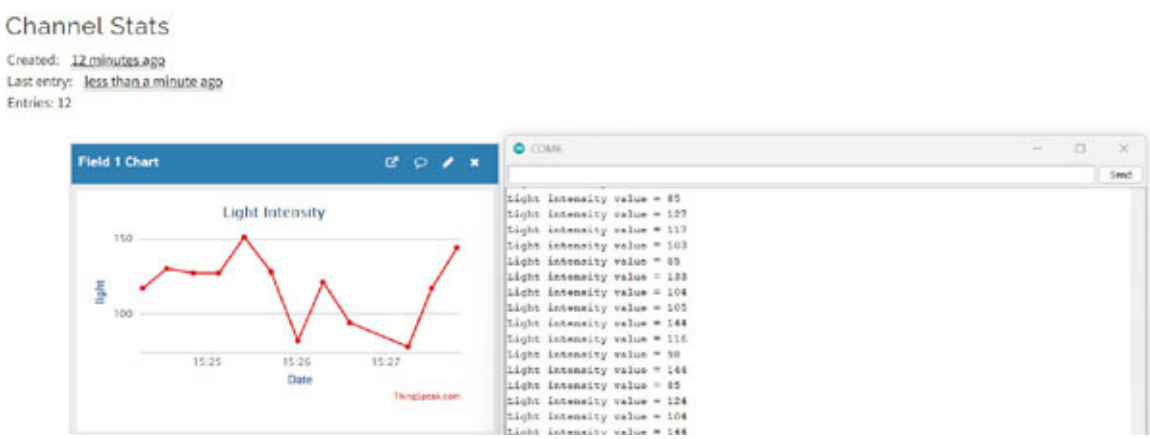


Figure 10.15: *Data visualized on the ThingSpeak server and serial monitor in real-time*

Recipe 5: Real-time Temperature and Humidity value logging on the web portal

In the previous recipe, we got a good idea of logging data on the web portals. We uploaded the light intensity value on the ThingSpeak server, and we can use this data anytime by exporting it back from the server. In this recipe, we will enhance our skills in the IoT domain and upload multiple sensor values on

a ThingSpeak channel. We will use the DHT22 temperature and humidity sensor to upload both temperature and humidity values to the ThingSpeak server.

The DHT series of sensors is very precise to measure temperature and humidity. DHT22 is an industry-standard sensor that can provide temperature and humidity values from a single sensor unit. The sensor and its pinout are shown in *Figure 10.16*. The sensor has four pins. This sensor works on the I2C protocol. Therefore, two pins are dedicated for ground and VCC, and the other two for data and clock.

Figure 10.16: *DHT22 sensor and its pinout*

The connections of the sensor with the ESP8266 development board are shown in *Figure 10.17*. We have not connected the clock pin of the sensor, but the data pin is connected with a pull-up resistor of 5kΩ.

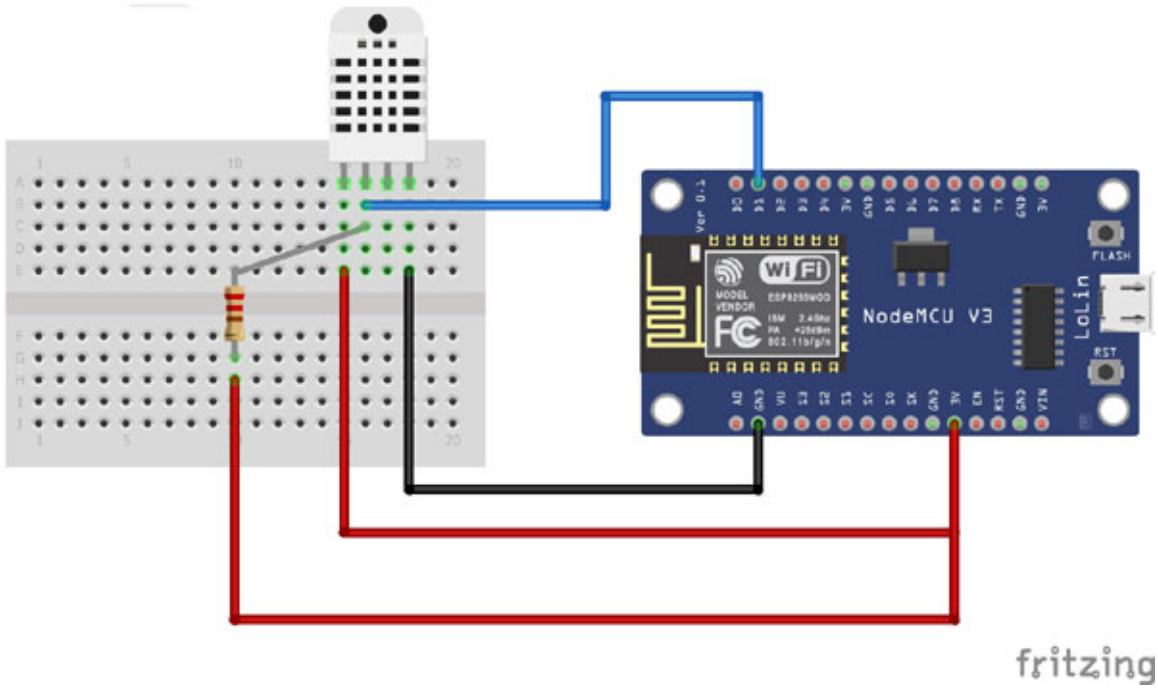


Figure 10.17: *Connections of the ESP8266 development board with the DHT22 sensor*

The sensor transmits the data serially. It sends temperature and humidity values one after the other. We have to configure the sensor to correctly identify it on the board. Therefore, we need to install the related libraries. Since this sensor is specifically used with ESP boards, we install the **`DHTesp`** library. We can also install the **`Adafruit unified Sensor`** library, although it is optional, to use all features of this sensor. The installations of both these libraries can be done through the library manager, as shown in *Figure 10.18*:

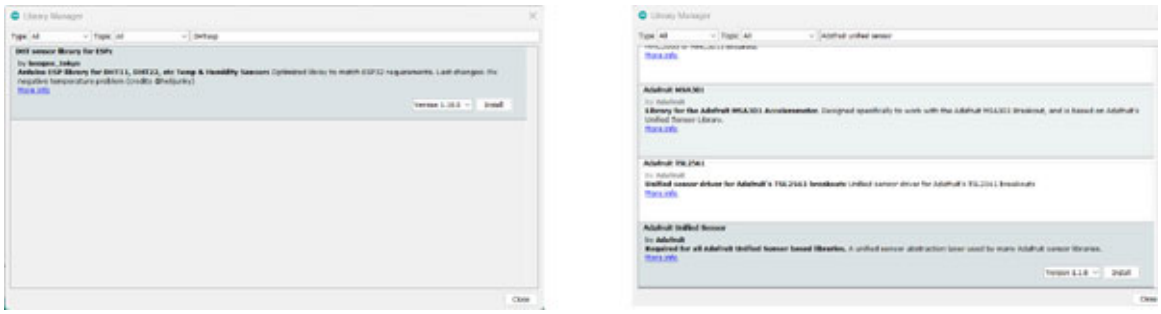


Figure 10.18: *Installing the related libraries for connecting the DHT22 sensor with ESP boards*

Once the hardware setup and related libraries are installed, we can write the code for this recipe. The code is as follows:

```
#include "DHTesp.h"
#include <ESP8266WiFi.h>
```

```
#include <WiFiClient.h>
#include <ThingSpeak.h>
const char* ssid = “OnePlusNord2”;
const char* password = “test1234”;
WiFiClient client;
unsigned long myChannelNumber = 1904970;
const char * myWriteAPIKey = “FSQGRDF5GL05244R”;
DHTesp dht;
void setup()
{
 Serial.begin(115200);
 Serial.println();
 WiFi.begin(ssid, password);
 ThingSpeak.begin(client);
 dht.setup(D1, DHTesp::DHT22);
}
void loop()
{
 delay(200);
 float humidity = dht.getHumidity();
 float temperature = dht.getTemperature();
 ThingSpeak.writeField(myChannelNumber, 1, temperature,
 myWriteAPIKey);
 delay(20000);
 ThingSpeak.writeField(myChannelNumber, 2, humidity,
 myWriteAPIKey);
 delay(20000);
}
```

In this code, we have called the **`DHTesp`** header file containing all the functions related to the DHT sensor. We defined the sensor as **`dht`** and used the **`DHT.setup`** function to define the pin and the type of sensor from the DHT series. Then, we used **`dht.getHumidity()`** and **`dht.getTemperature()`** to fetch the humidity and temperature values from the sensor via I2C communication. In the uploading process, we mentioned two different field

numbers to upload the temperature and humidity data, respectively. *Figure 10.19* shows the data values on the ThingSpeak server:

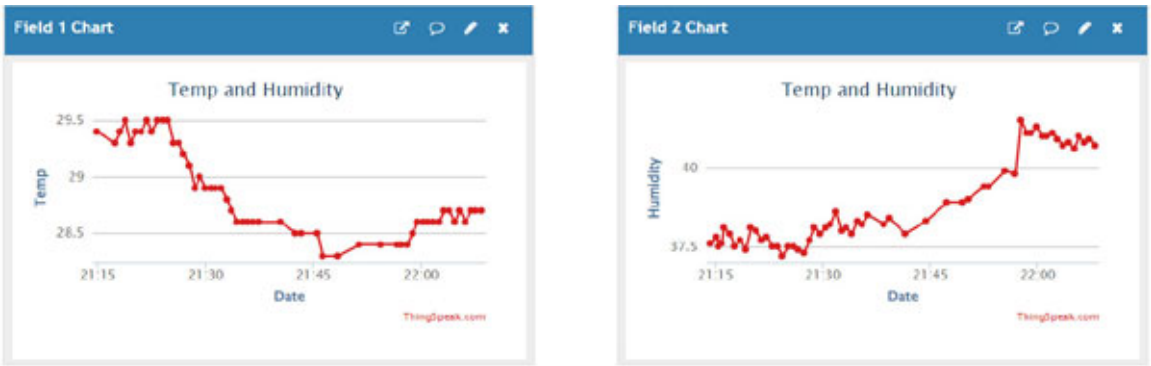


Figure 10.19: *Temperature and humidity values recorded from the DHT22 sensor*

Conclusion

In this chapter, we discussed a new wireless communication technique: Wi-Fi and connecting a microcontroller. We discussed that Arduino does not have the supported hardware connections to establish Wi-Fi communication; therefore, we used ESP8266 through the Arduino IDE for this purpose. We leant the process to configure the IDE for a new board and install the packages and libraries for the same. We discussed the connectivity process of ESP8266 with the Wi-Fi and related libraries. We also discussed the design of simple IoT applications, where we uploaded different sensor data to the web servers.

This chapter also concluded the book, and we hope you have learned various tools and techniques through these simple yet informative recipes. You can use them to design your own custom systems.

Points to remember

- The board package should be installed properly for any new board.
- The libraries for every new hardware component should be installed.
- Ensure that the correct ssid and password are selected to establish the Wi-Fi connection.
- Ensure that the correct channel number and API keys are mentioned to upload data on ThingSpeak.
- Ensure that the pull-up resistor is connected to the DHT22 pin.

Multiple choice questions

1. **How many physical pins are present on an ESP8266 board?**
    a. 16
    b. 30
    c. 26
    d. 20
2. **What is the full form of SSID in Wi-Fi communication?**
    a. Service Set Identifier
    b. Service Section Identity
    c. Short Service Identity
    d. Shared Service Identifier
3. **How many pins are present on the DHT22 sensor?**
    a. 1
    b. 2
    c. 3
    d. 4
4. **DHT22 sensor works on _____ communication.**
    a. Bluetooth
    b. SPI
    c. I2C
    d. parallel
5. **The BUILTIN LED on ESP8266 is attached to _____.**
    a. D0
    b. A0
    c. D2
    d. D13

Answers:

1. **b**
2. **a**
3. **d**
4. **c**
5. **a**

Index

Symbols

2-wheel drive (2WD) robot
designing 147, 148
2-wire interface (TWI) 172
4-digit password
door access, securing with 118, 119
6-level water level indicator
designing, LED array used 93, 94
7-segment LED display
driving 78-80
glowing patterns 80-84
16x2 LCD
blinking message, displaying 85-87
controlling, with I2C protocol 178-181
left scrolling message, displaying 87, 88

A

advanced programming constructs
5 LEDs, controlling with serial communication 63, 64
action, repeating fixed number of times 50, 51
action, repeating till another event occurs 51, 52
actions, performing based on variable values 46-48
binary values, sending through serial communication 59, 60
knight rider sequence, generating with 7 LEDs 62, 63
LED intensity, changing 58, 59
multiple actions, performing based on multiple values of variable 48-50
multiple decisions, making with case statements 60-62
ongoing event, interrupting with another event 55-57
text, splitting from group 54, 55
tones, generating with Arduino uno 57, 58
variables, grouping 52, 53
variables, splitting 52, 53
analog pins 34
Analog to Digital Converter (ADC) 34
analogWrite() function 36

Arduino 1, 2
Arduino board 2
  advantages 3
  connecting, to computer 8-10
  different boards 4
  sample code, uploading 10-12
Arduino IDE
  installing 5
  offline IDE installation, for desktop environments 5-8
  setting, for ESP32 or ESP8266 boards 235-238
Arduino pins
  accessing 32
  analog pins 34, 35
  digital input/output pins 33, 35
  power pins 33
Arduino programming
  constants 26
  data types 26, 27
  functions 24, 25
  pin levels 26
  variables 26
Arduino sketch
  creating 22
  library, accessing 14
  library, adding 14
  library, installing from library manager 15, 16
  library, installing from .zip file 17
  pre-installed library, accessing 15
  saving 22
  structuring 22, 23
  writing 23, 24
Arduino Uno R3 4
  features 4, 5
AT commands 217
automated parking boom barrier 160, 161
autoswitcher function 129

B

blinking message
  displaying, on 16x2 LCD 85-87

Bluetooth 212
Bluetooth controlled servo motor 228, 229
Bluetooth Low Energy (BLE) Module 212
Bluetooth modules
  Bluetooth connection, establishing with PC 215, 216
  connecting, to Windows PC 217, 218
  connecting, with Arduino 212
  HC-05 213, 214
  HC-06 214
  HC-10 214
  types 212-214
Boolean constants 26

C

CH-06 214
clap-controlled home appliance 105, 106
constants 26
  Boolean constants 26
Controller Area Network (CAN) 197
  using 198
  working principle 198-200
C programming
  basics 21

D

data
  displaying, on OLED screen 181-184
  types 26, 27
DC motor 139
  connecting, to Arduino 139
  controlling, with Arduino 145-147
  controlling, with encoder 166-169
  controlling, with L293D IC 142, 143
  direction, controlling 140, 141
  speedm controlling with PWM 144, 145
digital input/output pins 33
digital voltmeter design
  with Arduino 185-187
digital weather station

designing 128-132

F

flex sensor 162

H

HC-05 module 213, 214
HC-06 module 214
controlling, with AT commands 216, 217
HM-10 module 214

I

I2C adapter
connecting, to Arduino 176, 177
I2C communication 172
setting up, by connecting Arduino boards 172-175
Infrared (IR) proximity sensor 101
distance of wall, measuring 103, 104
object, detecting with 101, 102
Integrated Development Environment (IDE) 5
Inter Integrated Circuits (I2C) 39
Internet-of-Things (IoT) field 243
interrupt pins 40

K

knight rider circuit 62

L

L293D IC 142
LED array
6-level water level indicator, designing with 93, 94
LED blink
ESP8266 using 238-240
left scrolling message
displaying, on 16x2 LCD 87, 88
library, Arduino sketch
accessing 14
adding 14

installing, from library manager 15, 16
installing, from .zip file 17
pre-installed library, accessing 15
Library Manager 14
Light Dependent Resistor (LDR) 97, 99
Light Emitting Diode (LED) 69, 74
light intensity
detecting 98-100
line following robot
designing 151-153
Liquid Crystal Display (LCD) 14, 69
LM35 126
LPG leakage
detecting 111, 112

M

MCP2515 module
features 201-203
setting up, with Arduino 200, 201
mobile app configuration modes
controller mode 221
dimmer mode 221
switch mode 221
terminal mode 221
mobile phone
configuring for Arduino connection, via Bluetooth 219-223
multi-password door lock
designing 121, 122

N

NeoPixel LED
rotating sequence, displaying with 89-91
normally closed (NC) pin 146
normally open (NO) pin 146
number
converting, to string 40, 41

O

obstacle avoider robot

designing 148-150
offline IDE
installation, for desktop environments 5-8
Organic Light-Emitting Diodes (OLED) 69
data, displaying on 181-184

P

Parameter Group Number (PGN) 200
Passive Infrared Sensor (PIR) 97, 113
movement, detecting 113, 114
pin levels
HIGH 26
LOW 26
potentiometer 47
voltage values, generating with 91, 92
power pins 33
Pulse Width Modulation (PWM) 35
pushbutton 72
connections 72
time, measuring, when pressed 73, 74
PWM pins 35, 36
I2C pins 39
interrupt pins 40
serial communication pins 36-38
SPI pins 38, 39

R

real-time light intensity values
logging, on web 243-248
real-time temperature and humidity value
logging, on web 248-252
reverse car parking assistant system
designing 124
RGB LED
colors, generating 74, 75
random colors, generating 76, 77
RGB spectrum 76
robotic hand
designing 161-163

robust burglar alarm system
  designing 114-117
rotating sequence
  displaying, with NeoPixel LED 89-91
Rotations Per Minute (RPM) 166

S

sample code upload, on Arduino board
  compilation-related errors 13, 14
  performing 10-12
  related errors, uploading 14
  troubleshooting 12, 13
Serial Clock (SCL) 39
Serial Data (SDA) 39
serial monitor
  displaying on 27, 28
  printing, setting up 28, 29
  serial communication, setting up 28, 29
serial monitor information
  visualizing, over Bluetooth 223-225
Serial Peripheral Interface (SPI) 38
servo motor 154
  angle of movement, controlling 154, 155
  movements, controlling with potentiometer 156, 157
simple arithmetic operations
  addition 30
  performing 30
  working 31
smart dustbin
  designing 158, 159
smartphone controlled car 226, 227
Society of Automotive Engineers (SAE) 199
SPI.attachInterrupt(handler) function 189
SPI.begin() function 189
SPI communication
  for Arduino boards 189-193
  on Arduino board 187-189
SPI.setClockDivider(divider) function 189
SPI.transfer(val) function 189
stepper motor

controlling, with Arduino 164
string 40
number, converting to 40, 41
switch 70
voltage values, reading from 70-72

T

temperature 126
temperature-controlled alarm
with LM35 sensor 126, 127
temperature sensor data
sending, over CAN 203-208
ThingSpeak 244
traffic management 160

U

ultrasonic sensor-based water level indicator
designing 107-109
Universal Asynchronous Receiver Transmitter (UART) 37
Universal Synchronous Asynchronous Receiver Transmitter (USART) 37

V

variables 26
voltage values
generating, with potentiometer 91, 92

W

Wi-Fi
connection, establishing 233-235
ESP8266, connecting 241, 242