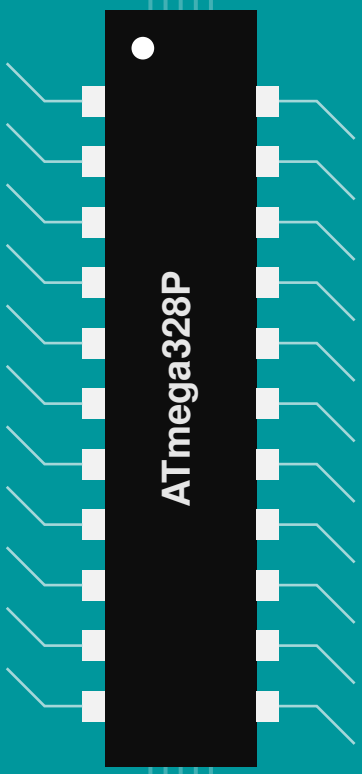




ARDUINO

Basics & Sample Codes

A Beginners Guide for Arduino Uno

SUMIT NARAYAN SAUROV



Ahsanullah University of Science and Technology

Department of Electrical and Electronic Engineering

Laboratory Manual

for

Electrical and Electronic Engineering Sessional Course

Student Name :

Student ID :

Course No. : EEE 3110

Course Title : Microcontroller and Microprocessor Laboratory

For the students of the

Department of Electrical and Electronic Engineering

3rd Year, 1st Semester

TABLE OF CONTENTS

Chapter 1 Serial Monitor 11

Chapter 2 Light Emitting Diode 41

Chapter 3 Switch 55

Chapter 4 Liquid Crystal Display 69

Chapter 5 Seven Segment Display 79

Chapter 6 Analog Operation 103

Chapter 7 Keypad 121

Chapter 8 SUART Communication 133

LIST OF FIGURES

Chapter 1

1.1 Serial Monitor Settings 12
1.2 Arduino Process Flowchart 15
1.3 Loop Process Flowchart 25

Chapter 2

2.1 LED Construction and Methodology 41
2.2 Arduino UNO Pinouts 42
2.3 LED Circuit Connection 46
2.4 External LED Connection with Arduino 47
2.5 Breadboard Connection of Figure 2.4 47
2.6 Red & Green LED Connection with Arduino 50
2.7 Breadboard Connection of Figure 2.6 50
2.8 Data Directional Register (DDR) 52
2.9 PORT Register (PORT) 52

Chapter 3

3.1 Different Types of Switches 55
3.2 Push Button Structure 56
3.3 Slide Switch Structure 56
3.4 Pull-Down vs Pull-Up Configuration 57
3.5 External Pull-Down 59
3.6 Breadboard Connection of Figure 3.5 59
3.7 External Pull-Up 61
3.8 Breadboard Connection of Figure 3.7 61
3.9 Internal Pull-Up 63
3.10 Breadboard Connection of Figure 3.9 63

3.11 Slide Switch . . . 65
3.12 Breadboard Connection of Figure 3.11 . . . 65
3.13 Logic Gate . . . 67
3.14 Breadboard Connection of Figure 3.13 . . . 67

Chapter 4

4.1 LCD Pinouts . . . 69
4.2 I2C LCD Pinouts . . . 70

Chapter 5

5.1 Seven Segment Display . . . 79
5.2 Internal Connection Diagram . . . 80
5.3 Pinouts . . . 81
5.4 Arduino Connection with 7SD . . . 82
5.5 Breadboard connection of Figure 5.4 . . . 82
5.6 HEX Digits . . . 84
5.7 4×7SD Pinouts . . . 94
5.8 4×7SD connection with Arduino . . . 95
5.9 4×7SD Internal Connection . . . 96
5.10 All Possible States . . . 96

Chapter 6

6.1 Reading Analog Voltage (3.3V) . . . 110
6.2 Potentiometer Voltage Reading . . . 110
6.3 PWM & Duty Cycle . . . 115
6.4 Analog Write (PWM) Operation . . . 116
6.5 Breadboard Connection of Figure 6.4 . . . 116
6.6 Dimmable LED Control using Potentiometer . . . 119

Chapter 7

7.1 4×4 Keypad Pins . . . 122
7.2 Keypad Connection with Arduino . . . 123

Chapter 8

8.1 UART Protocol . . . 134
8.2 SUART Connection between Two Arduino UNOs . . . 136

LIST OF TABLES

Chapter 1

1.1 Escape Sequence . . . 13
1.2 Data Types . . . 17
1.3 Arithmetic Operators . . . 18
1.4 Logical Operators . . . 19
1.5 Bitwise Operators . . . 19
1.6 Operator Precedence & Associativity Table . . . 30
1.7 Trigonometric Functions . . . 33
1.8 Exponential / Power / Logarithmic Functions . . . 33
1.9 Rounding Functions . . . 34
1.10 Range Related Functions . . . 34
1.11 Random Number Realted Functions . . . 34
1.12 Bit Level Function . . . 36
1.13 String Methods List . . . 38
1.14 Example of String Methods . . . 39
1.15 String-to-Number Conversion Methods . . . 39

Chapter 2

2.1 Digital IO Functions . . . 43
2.2 Digital Output . . . 44
2.3 Different Color LEDs . . . 45
2.4 Arduino UNO Ports . . . 51
2.5 Port Registers . . . 51
2.6 Configuration of Port Registers . . . 51

Chapter 3

3.1 Logic Gates . . . 68

Chapter 4

4.1 I2C LCD Connection with Arduino 70
4.2 Basic Methods 72
4.3 Display & Cursor Methods 73
4.4 Text Direction & Scrolling Methods 73

Chapter 5

5.1 Comparison Table 80
5.2 Connection between CC 7SD & Arduino 81
5.3 Common Cathode (CC) Code 87
5.4 Connection between 4×7SD & Arduino 94

Chapter 6

6.1 ADC Parameters 104
6.2 Analog Read related Functions 104
6.3 Reference Voltage 105
6.4 Reference Selection Bits 108
6.5 ADC Left Adjust Result Bit 109
6.6 Analog Channel Selection Bits 109
6.7 PWM in Arduino UNO 112
6.8 PWM Duty Cycle 115

Chapter 7

7.1 Keypad Connection with Arduino 123
7.2 Keypad Characters with ASCII Values 124
7.3 Keypad Methods 124

Chapter 8

8.1 SUART Connection 136

CODE LISTINGS

Chapter 1

1.1 Basic Printing . . . 12
1.2 Escape Sequence . . . 14
1.3 setup() vs loop() . . . 15
1.4 Number . . . 16
1.5 Variable . . . 17
1.6 Arithmetic Operator . . . 18
1.7 Logical Operators . . . 20
1.8 Bitwise Operators . . . 21
1.9 if - else if - else . . . 22
1.10 switch - case . . . 23
1.11 for Loop . . . 24
1.12 while Loop . . . 25
1.13 User Defined Custom Function . . . 26
1.14 Input from User . . . 28
1.15 Precedence & Associativity . . . 31
1.16 Type casting . . . 32
1.17 constrain() & map() . . . 35
1.18 Bit Level Functions . . . 36
1.19 Array . . . 37
1.20 String . . . 40

Chapter 2

2.1 Built-in LED Blink . . . 44
2.2 External LED Blink . . . 48
2.3 #define Directive . . . 48
2.4 'n' Times Blink . . . 49
2.5 Port Manipulation . . . 53
2.6 Bit Manipulation . . . 54

Chapter 3

3.1 External Pull-Down . . . 60
3.2 External Pull-Up . . . 62
3.3 Internal Pull-Up . . . 64
3.4 Slide Switch . . . 66
3.5 Logic Gate . . . 68

Chapter 4

4.1 I2C Address . . . 71
4.2 Hello World . . . 72
4.3 Set Cursor . . . 74
4.4 Formatted Data Display . . . 75
4.5 Serial Display . . . 76
4.6 Rotating Text . . . 77

Chapter 5

5.1 Basic . . . 83
5.2 Bit Read . . . 88
5.3 Array . . . 89
5.4 Loop . . . 90
5.5 Function . . . 91
5.6 Show All . . . 92
5.7 User Input . . . 93
5.8 4-Character Word Display . . . 98
5.9 4-Digit Decimal Number Display . . . 99
5.10 4-Digit Float Number Display . . . 101

Chapter 6

6.1 Analog Read . . . 111
6.2 Analog Write . . . 117
6.3 Fading LED . . . 118
6.4 Analog Read & Write . . . 120

Chapter 7

7.1 Basic Keypad 125
7.2 Numeric Keypad 126
7.3 Alphabetic Keypad 127
7.4 Simple Calculator 128
7.5 Fixed Length Number 131
7.6 Variable Length Number 132

Chapter 8

8.1T One Way Communication (Transmitter UNO) 137
8.1R One Way Communication (Receiver UNO) 138
8.2T Two Way Communication (UNO 1) 139
8.2R Two Way Communication (UNO 2) 140
8.3T Controlling Device (Transmitter UNO) 141
8.3R Controlling Device (Receiver UNO) 142

Chapter 1

Serial Monitor

The Serial Monitor of the Arduino Uno is a built-in communication tool in the Arduino IDE that allows real-time data exchange between the Arduino board and a computer through a USB connection. It operates using serial communication (UART), typically over the USB-to-serial interface provided by the ATmega16U2 on the Uno, enabling the microcontroller (ATmega328P) to send and receive data one byte at a time. The Serial Monitor is widely used for debugging, testing, and data visualization, as it allows programmers to display sensor readings, variable values, and status messages using functions. It also supports user input, where text or numeric data typed into the monitor can be transmitted to the Arduino, making it useful for interactive control and parameter tuning. Communication settings such as baud rate must match between the Serial Monitor and the program running on the Arduino to ensure reliable data transfer. Additionally, the Serial Monitor can handle different line-ending options (no line ending, newline, carriage return, or both), which is important when parsing incoming commands. Overall, the Serial Monitor serves as a simple yet powerful interface for observing program behavior, diagnosing errors, and facilitating two-way communication during development with the Arduino.

Serial Print

The `Serial.print()` function is used to transmit data from the Arduino to a computer through the UART serial communication interface. It sends data in human-readable ASCII format, making it especially useful for debugging, monitoring sensor values, and observing program execution using the Serial Monitor of the Arduino IDE. Before using `Serial.print()`, serial communication must be initialized with `Serial.begin(baudRate)`, where the baud rate (commonly 9600 bps) must match the Serial Monitor setting. The function supports printing of various data types including integers, floating-point numbers, characters, strings, and variables. Optional formatting allows numeric values to be displayed in different number bases such as decimal, binary, and hexadecimal, or with a specified number of decimal places for floating-point values. Unlike `Serial.println()`, `Serial.print()` does not automatically move the cursor to a new line after printing, which allows multiple values to be displayed on the same line in a controlled manner. Data is transmitted through the USB interface using the onboard USB-to-serial converter, enabling real-time monitoring of system behavior during program execution.

Code 1.1 Basic Printing

```
/*
 * In this code we will learn how to print something using Serial Monitor.
 * ---- (This is Multi-Line Comment Style) ----
 */

void setup() {

  // Initializing Serial Monitor (---- Single Line Comment Style ----)
  Serial.begin(9600); // Setting Baud Rate
  Serial.println(); // Avoids first line garbage characters

  // ------ Basic Print Operation ------ //
  Serial.print("1st line.");
  Serial.print("Same line.");

  // First prints text and then creates a New line.
  Serial.println("New line will be printed after this.");
  Serial.println("2nd line.");
  Serial.print("3rd line.");

}

void loop() {

}
```

Serial Monitor Settings

- Autoscroll On (⩔)
- No Line Ending (No automatic new line)
- 9600 baud (Baud rate = 9600)

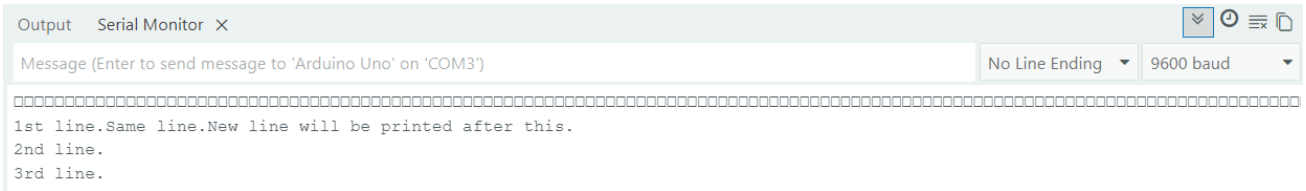


Figure 1.1: Serial Monitor Settings

Output

```
1st line.Same line.New line will be printed after this.
2nd line.
3rd line.
```

Escape Sequence

An escape sequence is a special combination of characters in programming that represents a character you cannot easily type or display directly, or represents a special non-printable character, or performs a specific control action within a string or character literal. It starts with a backslash (\), which tells the compiler or interpreter about special printing and escapes the normal meaning of the character that follows it. The backslash is followed by one or more characters, and together they are interpreted differently from their literal meaning by the compiler or interpreter. Escape sequences are commonly used to format output, control cursor position, and insert characters that cannot be typed directly using the keyboard. For example, `\n` inserts a newline, moving the cursor to the next line, while `\t` creates a horizontal tab space. Other escape sequences such as `\r` (carriage return) and `\b` (backspace) are used for fine control over text output. In addition, escape sequences allow the inclusion of special characters like quotation marks (`\"`), single quotes (`\'`), and the backslash itself (`\\`) inside strings without causing syntax errors. Some programming languages also support numeric escape sequences, such as octal or hexadecimal representations, to specify characters using their ASCII or Unicode values. Overall, escape sequences play a crucial role in text handling and formatted output, enabling programmers to precisely control how data is displayed or interpreted by the system. They also improve the readability and flexibility of programs by allowing complex text formatting within simple statements. Escape sequences are widely supported in many programming languages such as C, C++, Java, and Python, though their exact behavior may slightly vary. Understanding escape sequences helps programmers manage strings efficiently and avoid unintended interpretation of special characters. They are especially useful in file handling, console output, and data representation where precise control over character display is required. In embedded systems such as the Arduino platform, escape sequences are commonly used for serial communication and display formatting. For example, they help control how text appears on the Serial Monitor by managing line breaks and spacing. This allows clearer debugging messages and better interaction with sensors and external devices.

Table 1.1: Escape Sequence

Escape	Meaning	Effect in Serial Monitor
\n	New line	Moves cursor to next line
\r	Carriage return	Moves cursor to line start
\t	Tab	Adds horizontal space
\"	Double quote	Prints "
\'	Single quote	Prints '
\\	Backslash	Prints \

Code 1.2 Escape Sequence

```
/*
   In this code we will learn about Escape Sequence.
*/

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

  // -------- New Line -------- //
  Serial.print("Hello\nWorld");
  Serial.print("\n");


  // -------- Tab -------- //
  Serial.print("A\tB\tC\n");


  // -------- Double quote -------- //
  Serial.print("He said: \"Arduino\"\n");


  // -------- Backslash -------- //
  Serial.print("Path: C:\\Arduino\\Code\n");

  // Serial.println() automatically adds \r\n at the end
}

void loop() {

}
```

Output

```
Hello
World
A   B   C
He said: "Arduino"
Path: C:\Arduino\Code
```

Process Flow

In the Arduino program structure, execution follows a simple and continuous process flow: **Start → void setup() → void loop()**. When an Arduino board is powered on or reset, the program begins execution from the start and first runs the `setup()` function. This function is executed only once and is used to initialize system settings such as configuring input/output pins, starting serial communication, or setting up libraries and devices. After completing `setup()`, the program automatically enters the `loop()` function, which runs repeatedly in an infinite cycle as long as the board remains powered. The `loop()` function contains the main program logic, such as reading sensor data, processing inputs, and controlling outputs. This continuous execution enables Arduino systems to perform real-time monitoring and control tasks efficiently

without requiring manual restarts. This structure ensures that the system continuously responds to changing inputs and conditions in real time. It also simplifies program design by separating one-time initialization tasks from repeatedly executed operations.

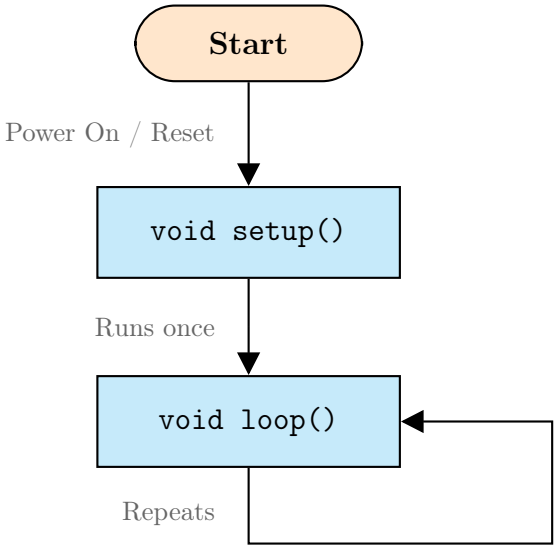


Figure 1.2: Arduino Process Flowchart

Code 1.3 setup() vs loop()

```
/*
 *  In this code we will learn the difference of two functions.
 *  ----- void setup() vs void loop() -----
 */

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial.println("This line will be printed just once.");
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.println("This Line will be printed multiple times.");
  // delay(1000);
}
```

Output

```
This line will be printed just once.
This Line will be printed multiple times.
This Line will be printed multiple times.
This Line will be printed multiple times.
.
.
.
```

Number System

In Arduino, the `Serial.print()` function allows printing numbers in different formats. By default, numbers are printed in **Decimal (base 10)**. Numbers can also be printed in **Binary (base 2)** using `BIN` or in **Hexadecimal (base 16)** using `HEX`. In addition to specifying the format in the function, Arduino also supports **number literals with prefixes**: `0d` for decimal, `0b` for binary, and `0x` for hexadecimal. These options provide flexibility for representing numbers in different bases while printing to the serial monitor.

Code 1.4 Number

```
/*
   In this code we will learn about Number System.
*/

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

  // Decimal Numbers
  Serial.println(12); // 12 (Decimal)
  Serial.println(18, DEC); // 18 (DEC --> optional)

  // Binary Numbers (*0b* prefix)
  Serial.println(0b101, BIN); // 101 (Binary Number)

  // HEX Numbers (*0x* prefix)
  Serial.println(0x12, HEX); // 12 (HEX Number)

  // Coversion
  Serial.println(0b101); // 5 (Binary to Decimal Conversion)
  Serial.println(0x12, DEC); // 18 (HEX to Decimal Conversion) [DEC --> optional]
  Serial.println(12, HEX); // C (Decimal to HEX Conversion)
  Serial.println(0xA, BIN); // 1010 (HEX to Binary Conversion)
  Serial.println(0b1111, HEX); // F (Binary to HEX Conversion)
}

void loop() {

}
```

Data Types

Data types define the kind of data a variable can store, such as integers, floating-point numbers, or characters, and determine how much memory is allocated for that data. Selecting an appropriate data type is essential for efficient memory usage and reliable program execution, especially in embedded systems with limited resources. Each data type provides a specific range of values and level of precision suited to different applications, such as sensor data processing, mathematical calculations, or control operations. Proper understanding of data types also helps prevent errors like overflow and improves overall program performance. The following table presents some of the most commonly used data types in Arduino UNO platform along with their key characteristics.

Table 1.2: Data Types

Type	Size	Range	Sign	Other Names
byte	8-bit	0 to 255	Unsigned	unsigned char
char	8-bit	−128 to 127	Signed	–
int	16-bit	−32,768 to 32,767	Signed	–
unsigned int	16-bit	0 to 65,535	Unsigned	word
long	32-bit	−2,147,483,648 to 2,147,483,647	Signed	–
unsigned long	32-bit	0 to 4,294,967,295	Unsigned	–
float	32-bit	6–7 decimal digits precision	Signed	double

Global vs Local Variable

In Arduino programming variables are commonly declared either globally or locally depending on their use. A global variable is usually declared at the top of the sketch, before the `setup()` and `loop()` functions, so it can be accessed by both functions and remains in memory throughout the program execution. In contrast, a local variable is declared inside `setup()` or `loop()` and can only be used within that specific function, existing only while the function runs. Global variables are used for values that must be shared between `setup()` and `loop()`, while local variables are used for temporary tasks within a function.

Code 1.5 Variable

```
/*
 * In this code we will learn about "Global vs Local" variables.
 */

int x = 5; // Global Variable

void setup() {
  Serial.begin(9600);
  Serial.println(x); // prints 5

  int y = 10; // Local Variable
  Serial.println(y); // prints 10
}

void loop() {
  Serial.println(x); // prints 5
}
```

Arithmatic Operators

Arithmetic operators are the bread and butter of Arduino programming, allowing your board to process data and make logic-based decisions. These operators work just like the math you learned in school, using standard symbols for addition, subtraction, multiplication, and division. A unique addition to this set is the modulo operator, which is incredibly useful for timing or looping tasks because it calculates the remainder of a division. Using these operators correctly is fundamental for tasks ranging from simple calculations to complex sensor data processing and controlling hardware based on numerical logic. Included below is a reference table of most commonly used arithmetic operators followed by a simple Arduino sketch demonstrating these basic arithmetic operations.

Table 1.3: Arithmetic Operators

Operator	Name	Example
$+$	Addition	$5 + 3 \rightarrow 8$
$-$	Subtraction	$10 - 4 \rightarrow 6$
$*$	Multiplication	$6 * 2 \rightarrow 12$
$/$	Division	$8 / 2 \rightarrow 4$
$\%$	Modulus	$7 \% 3 \rightarrow 1$

Code 1.6 Arithmetic Operator

```
/*
    In this code we will learn about basic Arithmetic Operators.
*/


void setup() {

  Serial.begin(9600);

  Serial.println(5 + 3); // Add (Result = 8)
  Serial.println(10 - 4); // Subtract (Result = 6)
  Serial.println(6 * 2); // Multiply (Result = 12)
  Serial.println(8 / 2); // Divide (Result = 4)
  Serial.println(7 % 3); // Remainder (Result = 1)
}

void loop() {

}
```

Logical Operators

Logical operators in Arduino are used to combine or modify relational expressions and make decisions based on multiple conditions. These operators evaluate expressions and return a boolean result, either `true` (1) or `false` (0). Logical operators are commonly used in control structures to implement decision-making logic in embedded systems. Proper use of logical operators improves program efficiency and allows the microcontroller to respond intelligently to sensor inputs, user commands, and system states during real-time operation.

Table 1.4: Logical Operators

Operator	Name	Description
&&	Logical AND	True if both conditions are true
||	Logical OR	True if at least one condition is true
!	Logical NOT	Reverses the logical state of a condition

Bitwise operators

Bitwise operators are special operators that perform operations directly on the individual bits of integer data types. These operators allow efficient manipulation of data at the binary level, which is particularly useful in embedded programming, hardware control, and memory-constrained applications. Bitwise operators are also essential in setting, clearing, or checking the status of individual pins on Arduino digital I/O ports, allowing fine-grained control of hardware.

Table 1.5: Bitwise Operators

Operator	Name	Description
&	Bitwise AND	1 if both corresponding bits are 1
|	Bitwise OR	1 if at least one corresponding bit is 1
$\wedge$	Bitwise XOR	1 if corresponding bits are different
$\sim$	Bitwise NOT	Inverts all bits
$\ll$	Left shift	Shifts bits to the left, adding 0 at right
$\gg$	Right shift	Shifts bits to the right, adding 0 at left

Code 1.7 Logical Operators

```
/*
   In this code we will learn about Logical Operators.
        - Logical AND (&&)
        - Logical OR (||)
        - Logical NOT (!)
*/

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

  int a = 6;
  int b = 3;

  // Logical AND (&&)
  Serial.println((a > 5) && (b < 5)); // 1 (true)

  // Logical OR (||)
  Serial.println((a > 10) || (b < 5)); // 1 (true)

  // Logical NOT (!)
  Serial.println(!(a > 5)); // 0 (false)
}

void loop() {

}
```

Output

```
1
1
0
```

Logical vs Bitwise Operator

In programming, understanding the distinction between logical and bitwise operators is crucial because they operate at fundamentally different levels. Logical operators evaluate entire Boolean expressions and treat their operands as whole entities — usually considering whether the value is "true" (nonzero) or "false" (zero). For example, when using &&, the program checks whether both expressions are true as complete values, and it may employ short-circuit evaluation, meaning it stops checking as soon as the outcome is determined. Bitwise operators, on the other hand, work at the level of individual bits within a byte or integer. These operators manipulate each bit independently, allowing precise control over the binary representation of data. This means a single bit pattern in a number can be isolated, modified, or combined without affecting other bits, which is essential in low-level programming, embedded systems, or hardware interfacing. In essence, logical operators evaluate truthiness of entire values, while bitwise operators dissect and operate on the binary structure itself, giving the programmer fine-grained control over data. Recognizing this distinction helps prevent unintended behavior, especially when mixing these operators in conditional statements or arithmetic expressions.

Code 1.8 Bitwise Operators

```
/*
  In this code we will learn about different Bitwise operators.
*/

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

  byte a = 0b101;  // 5 in decimal
  byte b = 0b011;  // 3 in decimal

  // Bitwise AND
  Serial.print("a & b = ");
  Serial.println(a & b, BIN); // 001

  // Bitwise OR
  Serial.print("a | b = ");
  Serial.println(a | b, BIN); // 111

  // Bitwise XOR
  Serial.print("a ^ b = ");
  Serial.println(a ^ b, BIN); // 110

  // Bitwise NOT (single operand)
  Serial.print("~a = ");
  byte c = ~a;
  Serial.println(c, BIN);

  // Left shift
  Serial.print("a << 1 = ");
  Serial.println(a << 1, BIN); // 1010

  // Right shift
  Serial.print("a >> 1 = ");
  Serial.println(a >> 1, BIN); // 010
}

void loop() {

}
```

Output

```
a & b = 1
a | b = 111
a ^ b = 110
~a = 11111010
a << 1 = 1010
a >> 1 = 10
```

Selection Statements

Selection statements are control structures in Arduino programming that allow a program to choose different execution paths based on the evaluation of one or more conditions. These statements enable decision-making capability in a program, allowing the microcontroller to respond differently under varying input conditions, sensor readings, or system states. A condition is typically formed using relational and logical operators and is evaluated as either true or false. Depending on the result, specific blocks of code are executed while others are skipped. Selection statements improve program flexibility, efficiency, and readability by organizing logic into meaningful branches rather than relying on sequential execution only. They are widely used in embedded systems for implementing control logic, safety checks, user interaction, and automated responses in real-time applications.

`if-else if-else` statements are used to make decisions based on multiple conditions. The program first checks a condition, and if it evaluates to true, a specific block of code is executed. If the first condition is false, other conditions are checked sequentially using `else if` statements. If none of the conditions are satisfied, the code in the `else` block runs. This structure is useful when several possible cases need to be evaluated in order.

Code 1.9 if - else if - else

```
/*
   In this code we will learn about "if-else if-else" statement
   We will find out whether a number is positive, negative or zero
*/

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

  int num = 5;   // change this number to test

  if (num > 0) {
    Serial.println("Positive");
  }
  else if (num < 0) {
    Serial.println("Negative");
  }
  else {
    Serial.println("Zero");
  }
}

void loop() {

}
```

Output

```
Positive
```

For situations with multiple discrete options, `switch-case` statements provide a cleaner and more organized approach. Here, the program evaluates an expression and executes the matching case block. Each case ends with a `break` to prevent fall-through, and an optional `default` case can handle unexpected values. `switch-case` is often preferred when dealing with inputs like keypad entries or menu selections.

Code 1.10 switch - case

```
/*
   In this code we will learn about switch-case
   Depending on numeric day value it displays the corresponding day of the week
*/

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

  int day = 3;   // change this value (1-7)

  switch (day) {
    case 1:
      Serial.println("Sunday");
      break;

    case 2:
      Serial.println("Monday");
      break;

    case 3:
      Serial.println("Tuesday");
      break;

    case 4:
      Serial.println("Wednesday");
      break;

    case 5:
      Serial.println("Thursday");
      break;

    case 6:
      Serial.println("Friday");
      break;

    case 7:
      Serial.println("Saturday");
      break;

    default:
      Serial.println("Invalid");
  }
}

void loop() {
}
```

Loop Statements

Loop statements are control structures in Arduino programming that allow a block of code to be executed repeatedly as long as a specified condition remains true or for a predetermined number of iterations. They eliminate the need to write the same instructions multiple times, thereby reducing program size, improving readability, and simplifying program maintenance. A loop typically consists of an initialization step, a condition check, and an update operation that controls the repetition process. Loop statements are essential for tasks such as continuous sensor monitoring, data acquisition, signal generation, time-based operations, and real-time control in embedded systems.

The `for()` loop is used when the number of iterations is known in advance. It has three parts: initialization, condition, and increment/decrement. The loop executes repeatedly as long as the condition is true, making it ideal for tasks like counting, iterating through arrays, or performing repeated actions a fixed number of times.

Code 1.11 for Loop

```
/*
   In this code we will learn about "for" loop.

   It calculates the sum of numbers from 1 to 5.
   Sum = 1 + 2 + 3 + 4 + 5
*/

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

  int sum = 0;

  for (int i = 1; i <= 5; i++)
  {
    sum = sum + i;
  }

  Serial.print("Sum = ");
  Serial.println(sum);
}

void loop() {

}
```

Output

```
Sum = 15
```

The `while()` loop is used when the number of iterations is not known beforehand. It checks a condition before each iteration and continues looping as long as the condition remains true. This makes it suitable for scenarios where repetition depends on dynamic factors.

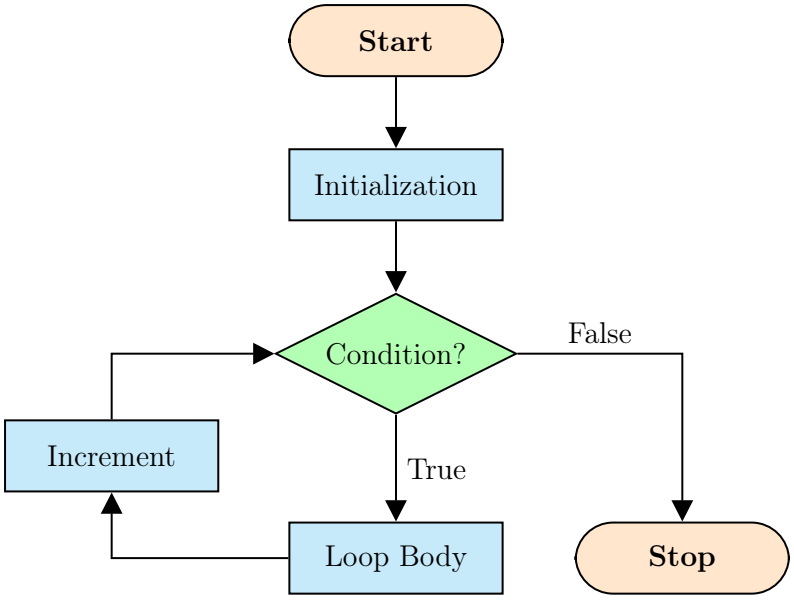


Figure 1.3: Loop Process Flowchart

Code 1.12 while Loop

```
/*
   In this code we will learn about "while" loop.

   It calculates the product of numbers from 1 to 5.
   Product = 1 * 2 * 3 * 4 * 5
*/

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

  int product = 1;
  int i = 1;

  while (i <= 5)
  {
    product = product * i;
    i++;
  }

  Serial.print("Product = ");
  Serial.println(product);
}

void loop() {

}
```

Output

```
Product = 120
```

User-Defined (Custom) Functions in Arduino Uno

User-defined or custom functions in Arduino Uno allow programmers to divide a large program into smaller, reusable, and logically organized blocks of code. A function performs a specific task and can be called multiple times from different parts of the program, which improves code readability, reduces repetition, and simplifies debugging and maintenance. A custom function typically consists of a function name, a return data type, optional parameters, and a function body containing executable statements. Functions may return a value using the `return` statement or may operate without returning any value using the `void` keyword. Parameters allow data to be passed into a function, enabling flexible and modular design. In embedded systems, user-defined functions are commonly used to handle repeated operations such as sensor reading, data processing, motor control, and communication tasks, making the program more structured, efficient, and scalable.

Code 1.13 User Defined Custom Function

```
/* In this code we will learn about "User Defined Function".
   This type of function is also called *Custom Function*.

   We will define a function which takes two integers as inputs
   and simply adds them and then returns the summation result.
*/

void setup() {

  Serial.begin(9600);

  // Function Call
  int s = summation(5, 10);
  Serial.print("Sum = "); // prints text "Sum = "
  Serial.print(s); // prints the value stored in variable *s* (15)
}

void loop()
{

}

// Custom Function Declaration
int summation(int a, int b)
{
  int sum;
  sum = a + b; // a = 5 & b = 10
  return sum;
}
```

Output

```
Sum = 15
```

Taking User Input in Arduino Uno via Serial Communication

User input in Arduino Uno is commonly taken through the Serial Monitor of the Arduino IDE using USB-based serial communication. Before receiving any data, serial communication must be initialized using `Serial.begin(baudRate)`, and the baud rate must match the Serial Monitor setting. The user can type characters, numbers, or text into the Serial Monitor input field and transmit the data by pressing the send button or the Enter key. Incoming data is stored temporarily in the serial receive buffer of the microcontroller and can be accessed using different serial input functions depending on the required data type and format. We mainly use three functions under Serial Streams to read String & Numbers – Integers and Floats.

- Serial.readString()
- Serial.parseInt()
- Serial.parseFloat()

Serial.readString()

The function `Serial.readString()` reads all available incoming characters from the buffer and stores them as a String object until a timeout occurs or a line termination character is received. This method is suitable for receiving full text commands or sentences but may consume more memory due to dynamic string allocation.

Serial.parseInt()

The function `Serial.parseInt()` reads characters from the serial buffer and automatically extracts the first valid integer value by ignoring non-numeric characters, stopping when a timeout or a non-digit character is encountered, making it useful for receiving numerical commands such as menu selections or parameter values.

Serial.parseFloat()

The function `Serial.parseFloat()` reads and converts incoming serial data into a floating-point number, supporting decimal values and optional sign characters, which is ideal for receiving sensor calibration values or real-time control parameters.

These parsing functions rely on a timeout mechanism defined by `Serial.setTimeout()`, which determines how long the system waits for incoming data before terminating the read operation. Proper handling of serial input also requires checking data availability using `Serial.available()` to avoid reading empty buffers and ensuring synchronization between transmitted and received data. Serial-based user input enables interactive control, debugging, and real-time parameter adjustment in embedded applications using Arduino Uno.

Code 1.14 Input from User

```
/*
   In this code we will learn how to take inputs from user.
   We will learn about these 3 types - (1) String (2) Integer (3) Float

   String input: Name
   Integer input: Student ID
   Float input: CGPA
*/

void setup() {

  Serial.begin(9600);


  // ----- String Input -----
  Serial.print("Enter your Name: ");
  while (Serial.available() == 0)
  {
    // Wait as long as there is no data available in Serial Buffer
    // That means user has not given any input from Serial Monitor
  }
  String n = Serial.readString();
  Serial.println(n);


  // ----- Integer Input -----
  Serial.print("Enter your ID: ");
  while (Serial.available() == 0) {}
  int id = Serial.parseInt(); // up to 32,767
  // unsigned long id = Serial.parseInt(); // up to 4,294,967,295
  Serial.println(id);


  // ----- Float Input -----
  Serial.print("Enter your CGPA: ");
  while (!Serial.available());
  float cg = Serial.parseFloat();
  Serial.println(cg); // default prints up to 2 decimal places (rounding)
  // Serial.println(cg,3); // prints 3 decimal places
}

void loop() {

}
```

Output

```
Enter your Name: Arif
Enter your ID: 9821
Enter your CGPA: 3.65
```

Operator Precedence

Operator Precedence determines the specific order in which different types of operators are evaluated within an expression, functioning similarly to the "PEMDAS" (Parentheses, Exponents, Multiplication, Division, Addition, Subtraction) rule used in standard mathematics. In the context of an Arduino sketch, this hierarchy is far more expansive, covering bitwise, relational, and logical operators. For instance, multiplication and division always hold higher precedence than addition and subtraction. Consider the expression `int result = 5 + 10 * 2;`. The Arduino will first evaluate 10×2 to get 20, and then add 5, resulting in 25. If the compiler read strictly from left to right without precedence, the result would be 30, potentially causing a critical error in a sensor scaling calculation. This hierarchy becomes even more vital when dealing with Boolean logic. For example, the logical AND (&&) has higher precedence than logical OR (||). Without this distinction, complex conditional statements — such as those checking multiple safety interlocks on a CNC machine or a 3D printer — could evaluate to `true` when they should be `false`, leading to hazardous hardware behavior.

Operator Associativity

Operator Associativity becomes the deciding factor when an expression contains multiple operators that share the exact same precedence level. While precedence tells the compiler which "group" of operators to handle first, associativity defines the direction — either left-to-right or right-to-left — in which those tied operators are processed. Most common arithmetic operators, such as addition (+), subtraction (−), multiplication (∗), and division (/), follow left-to-right associativity. This means in an expression like `100 / 5 * 2`, the division is performed first (20), followed by the multiplication, resulting in 40. However, certain operators, most notably the assignment operator (=) and compound assignments (like + = or ∗ =), utilize right-to-left associativity. For example, in the statement `a = b = 10;`, the Uno first evaluates the rightmost assignment, giving the value 10 to variable `b`, and then passes that result to variable `a`. Understanding this "direction of flow" is critical when chaining operations or manipulating registers at the bit level. Without these fixed rules of associativity, the compiler would lack a consistent starting point, rendering complex mathematical formulas and multi-variable initializations unpredictable and nearly impossible to debug.

Type Casting

Type Casting is used to convert one data type into another to ensure correct calculations and efficient memory usage. It is especially important when working with different data types such as `int`, `float`, `char`, and `byte`, as operations between mismatched types may lead to unintended results like loss of precision or integer division. Type casting can be either **implicit** or **explicit**. Implicit type casting (also called automatic type conversion) is performed automatically by the compiler when compatible data types are used in an expression, usually converting a lower precision type to a higher precision type. On the other hand, explicit type casting (also called manual type conversion) is specified by the programmer using a cast operator such as `(float)x`, allowing precise control over how values are treated during arithmetic or logical operations. Type casting is commonly applied in Arduino programs when handling sensor data, performing mathematical computations, or converting character data to numerical values, helping improve accuracy and program reliability in embedded system applications.

The following table provides a comprehensive reference for the hierarchy of Arduino (C++) operators. It organizes them from the highest precedence level at the top to the lowest at the bottom, while specifying the associativity for each group to clarify the evaluation flow.

Table 1.6: Operator Precedence & Associativity Table

Level	**Operators**	**Associativity**	**Description**
1 (Highest)	(.) [.] . ->	Left-to-Right	Function call, indexing, member access
2	postfix ++ −−	Left-to-Right	Postfix increment or decrement
3	prefix ++ −− +− (unary) ! ˜ ∗ (type)	Right-to-Left	Unary operators
4	∗ / %	Left-to-Right	Multiplicative
5	+ −	Left-to-Right	Additive
6	<< >>	Left-to-Right	Bitwise shifts
7	< <= > >=	Left-to-Right	Relational
8	== ! =	Left-to-Right	Equality
9	&	Left-to-Right	Bitwise AND
10	∧	Left-to-Right	Bitwise XOR
11	|	Left-to-Right	Bitwise OR
12	&&	Left-to-Right	Logical AND
13	||	Left-to-Right	Logical OR
14	? :	Right-to-Left	Ternary conditional
15	= += -= *= /= %= <<= >>= &= ˆ= |=	Right-to-Left	Assignment
16 (Lowest)	,	Left-to-Right	Comma operator

Code 1.15 Precedence & Associativity

```
/*
   In this code we will Learn about Operator Precedence & Associativity.
*/


void setup() {

  Serial.begin(9600);


  // -------- Precedence -------- //

  int x = 3 + 4 * 2 > 10 && 5 < 6;
  Serial.print("x = ");
  Serial.println(x);   // Output will be 1

  // Step 1: 4 * 2 is evaluated first → 8
  // Step 2: 3 + 8 is evaluated next → 11
  // Step 3: 11 > 10 → true (1)
  // Step 4: 5 < 6 → true (1)
  // Step 5: true && true → true (1)


  // -------- Associativity -------- //

  int y = 2 * 2 / 4;
  Serial.print("y = ");
  Serial.println(y);   // Output will be 1

  // Step 1: 2 * 2 = 4
  // Step 2: 4 / 4 = 1


  int z = 2 / 4 * 2;
  Serial.print("z = ");
  Serial.println(z);   // Output will be 0

  // Step 1: 2 / 4 = 0 [As 0.5 becomes 0 (integer division)]
  // Step 2: 0 * 2 = 0
}

void loop() {

}
```

Output

```
x = 1
y = 1
z = 0
```

As shown in the previous code, unexpected results may occur in arithmetic expressions due to operator precedence, associativity, or limitations of the data types used in calculations. When operations involve different data types or when precision is important, implicit evaluation may lead to loss of information or incorrect outcomes. This issue can be addressed using type casting, which explicitly converts operands to a desired data type before performing the operation. By controlling how values are treated during computation, type casting helps ensure accurate results and proper evaluation of expressions as demonstrated in the code below.

Code 1.16 Type casting

```
/* In this code we will Learn about Type Casting. */

void setup() {

  Serial.begin(9600);

  // --- Float → Int ---
  float a = 7.89;
  int b = (int)a;  // truncate decimal (Explicit Conversion)
  Serial.print("b = ");
  Serial.println(b);  // 7

  // --- Int → Float ---
  int c = 10;
  float d = c;  // Implicit Conversion
  Serial.print("d = ");
  Serial.println(d);  // 10.00


  // --- Use Type Casting to solve Associativity Issue --- //

  // Explicit Type Casting (Method 1)
  int x = (float) 2 / 4 * 2;
  Serial.print("x = ");
  Serial.println(x); // x = 1

  // Float Literal (Method 2)
  int y = 2.0 / 4 * 2;
  Serial.print("y = ");
  Serial.println(y); // y = 1
}

void loop() {

}
```

Output

```
b = 7
d = 10.00
x = 1
y = 1
```

Built-in Functions

In Arduino Uno many built-in functions available in C++ can be directly used without including any custom library. List of different functions by types are included here.

- ❒ Trigonometric Functions
- ❒ Exponential / Power / Logarithmic Functions
- ❒ Rounding Functions
- ❒ Range Related Functions
- ❒ Random Number Realted Functions

Table 1.7: Trigonometric Functions

Function	Description	Example
sin(x)	Sine of x (radians)	sin(PI/2) = 1
cos(x)	Cosine of x (radians)	cos(PI) = -1
tan(x)	Tangent of x (radians)	tan(PI/4) = 1
atan(x)	Arctangent of x	atan(1) = PI/4
atan2(y, x)	Arctangent of y/x considering quadrant	atan2(1,1) = PI/4

Table 1.8: Exponential / Power / Logarithmic Functions

Function	Description	Example
pow(base, exponent)	Base raised to exponent	pow(2,3) = 8
sqrt(x)	Square root of x	sqrt(9) = 3
sq(x)	Square of x	sq(3) = 9
log(x)	Natural logarithm (base e)	log(E) = 1
log10(x)	Base-10 logarithm	log10(100) = 2
exp(x)	e raised to power x	exp(1) = 2.718

Table 1.9: Rounding Functions

Function	Description	Example
ceil(x)	Smallest integer greater than or equal to x	ceil(2.3) = 3
floor(x)	Largest integer less than or equal to x	floor(2.7) = 2
round(x)	Round x to nearest integer	round(2.5) = 3, round(2.4) = 2
trunc(x)	Truncate decimal part	trunc(2.9) = 2, trunc(-2.9) = -2

Table 1.10: Range Related Functions

Function	Description	Example
abs(x)	Absolute value of x	abs(-5) = 5
constrain(x, a, b)	Constrain x between a and b	constrain(15, 0, 10) = 10
map(value, fromLow, fromHigh, toLow, toHigh)	Map value from one range to another	map(50, 0, 100, 0, 255) = 127
min(a, b)	Minimum of two values	min(5, 10) = 5
max(a, b)	Maximum of two values	max(5, 10) = 10

Table 1.11: Random Number Realted Functions

Function	Description	Example
random(max)	Random integer from 0 to max-1	random(10) = 0..9
random(min, max)	Random integer between min and max-1	random(5, 10) = 5..9
randomSeed(seed)	Initialize random number generator	randomSeed(1234)

constrain()

This function is used to limit a value within a specific range. It takes three arguments: the value to constrain, a minimum limit, and a maximum limit. If the value is lower than the minimum, the function returns the minimum; if it is higher than the maximum, it returns the maximum. If the value already falls within the specified range, it remains unchanged. This function is commonly used to keep sensor readings or calculations within safe and defined limits.

Code 1.17 constrain() & map()

```
/*
   In this code we will learn about two basic Math functions
        - contrain()
        - map()
*/

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

  // --------------- constrain() --------------- //
  // constrain(x, a, b) Constrain x between a and b

  Serial.println(constrain(15, 10, 20)); // 15
  Serial.println(constrain(5, 10, 20)); // 10
  Serial.println(constrain(25, 10, 20)); // 20



  // --------------- map() --------------- //
  // map(value, fromLow, fromHigh, toLow, toHigh)
  // map value from one range [fromLow fromHigh] to another range [toLow toHigh]
  // map value = 5 from [0 10] range to [0 100] range

  Serial.println(map(5, 0, 10, 0, 100)); // 50

}

void loop() {

}
```

map()

This function is used to re-scale a number from one range to another. It takes five arguments: the value to map, the lower and upper bounds of the input range, and the lower and upper bounds of the output range. It then calculates a proportional value within the new range based on the relative position of the input value. This function is commonly used to convert sensor readings from their original range to a different, desired output range.

Bit Manipulation

Bit manipulation refers to directly accessing and modifying individual bits of a data value. It allows efficient control of hardware and memory by operating at the lowest level of data representation. Bit-level functions enable programmers to check, modify, or update specific bits without affecting the entire variable. This approach is widely used in embedded systems for tasks like controlling digital pins, configuring registers, and optimizing memory usage, making bit manipulation an essential technique for efficient and precise hardware programming.

Table 1.12: Bit Level Function

Function Name	Description
`bitRead(variable, bit)`	Reads the value of the specified bit
`bitSet(variable, bit)`	Sets the specified bit to logic 1
`bitClear(variable, bit)`	Clears the specified bit to logic 0
`bitWrite(variable, bit, value)`	Writes the given value to the specified bit

Code 1.18 Bit Level Functions

```
/* In this code we will learn about different bit level functions */

void setup() {
  Serial.begin(9600);
  byte x = 0b101; // 5 (Decimal) = 00000101 (Binary)

  Serial.println(bitRead(x, 0)); // 1
  Serial.println(bitRead(x, 1)); // 0
  Serial.println(bitRead(x, 2)); // 1
  Serial.println(bitRead(x, 5)); // 0

  byte a = bitSet(x, 1);
  Serial.println(a, BIN); // 111

  byte b = bitClear(a, 0);
  Serial.println(b, BIN); // 110

  byte c = bitWrite(b, 1, 0);
  Serial.println(c, BIN); // 100
  byte d = bitWrite(c, 0, 1);
  Serial.println(d, BIN); // 101
}

void loop() {
}
```

Arrays

An array is a data structure that allows multiple values of the same data type to be stored under a single variable name using indexed memory locations. Each element of an array can be accessed individually using its index number, where indexing starts from zero. Arrays enable efficient handling of large sets of related data, such as sensor readings, lookup tables, and waveform samples, without the need to declare multiple separate variables. The size of an array must be defined at the time of declaration and remains fixed during program execution. Arrays can store different data types such as integers, floating-point values, characters, and strings. In embedded systems, arrays improve memory organization, simplify data processing using loops, and enhance program efficiency when performing repetitive operations like data acquisition, sorting, and signal generation. The following program demonstrates the basic use of an array. To determine the total number of elements in the array automatically, the `sizeof()` operator is used. By dividing the total size of the array by the size of a single element, the program calculates the array length dynamically. This approach is useful because it allows the program to adapt automatically if the array size is changed.

Code 1.19 Array

```
/*
   In this code we will learn about basic Array operation.
*/

int numbers[] = {10, 20, 30, 40, 50};

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

  int length = sizeof(numbers) / sizeof(numbers[0]);  // array length

  for (int i = 0; i < length; i++) {
    Serial.print("Array element ");
    Serial.print("[");
    Serial.print(i);
    Serial.print("]");
    Serial.print(" = ");
    Serial.println(numbers[i]);
  }
}

void loop() {

}
```

Output

```
Array element [0] = 10
Array element [1] = 20
Array element [2] = 30
Array element [3] = 40
Array element [4] = 50
```

String

The `String` class in Arduino provides a convenient way to create and manipulate text data dynamically without manually handling character arrays. A String object can store sequences of characters and supports various operations such as concatenation, comparison, sub-string extraction, searching, and conversion between numeric and text formats. Unlike traditional C-style character arrays, String objects automatically manage memory allocation and resizing, making string handling simpler and more readable for beginners. In Arduino applications, String is commonly used for displaying messages on the Serial Monitor, handling user input, formatting sensor data, and building communication packets. Some of the most commonly used String methods and their descriptions are presented in the following first table, while the corresponding usage examples for each method are provided in the second table for better understanding.

Table 1.13: String Methods List

Method	Description
`.length()`	Returns the number of characters in the string.
`.charAt(index)`	Returns the character at the given index (0-based).
`.substring(start,end)`	Extracts a part of the string; `start` inclusive, `end` exclusive.
`.indexOf(str)`	Returns the position of the first occurrence of a substring.
`.lastIndexOf(str)`	Returns the position of the last occurrence of a substring.
`.equals(str)`	Checks if two strings are exactly equal.
`.equalsIgnoreCase(str)`	Compares strings ignoring case.
`.toUpperCase()`	Converts the string to uppercase.
`.toLowerCase()`	Converts the string to lowercase.
`.concat(str)`	Appends another string to the end.
`+` operator	Strings can be concatenated using `+`.
`.remove(start,length)`	Removes part of the string.
`.replace(target,replace)`	Replaces all occurrences of a substring.
`.trim()`	Removes whitespace from both ends.

Table 1.14: Example of String Methods

Method	Example
`.length()`	`"Hello".length()` → 5
`.charAt(index)`	`"Hello".charAt(0)` → 'H'
`.substring(start, end)`	`"Hello".substring(0, 3)` → "Hel"
`.indexOf(str)`	`"Hello".indexOf("l")` → 2
`.lastIndexOf(str)`	`"Hello".lastIndexOf("l")` → 3
`.equals(str)`	`"Hello".equals("hello")` → false
`.equalsIgnoreCase(str)`	`"Hello".equalsIgnoreCase("hello")` → true
`.toUpperCase()`	`"Hello".toUpperCase()` → "HELLO"
`.toLowerCase()`	`"Hello".toLowerCase()` → "hello"
`.concat(str)`	`"Hi ".concat("There")` → "Hi There"
`+ operator`	`"Hi " + "There"` → "Hi There"
`.remove(start, length)`	`"Hello".remove(1, 3)` → "Ho"
`.replace(target, replacement)`	`"Hello".replace("l","x")` → "Hexxo"
`.trim()`	`"  Hi  ".trim()` → "Hi"

The following table provides examples of the String methods used for numerical conversion.

Table 1.15: String-to-Number Conversion Methods

Method	Description	Example
`.toInt()`	Converts a string into an integer	`"1234".toInt()` → 1234
`.toFloat()`	Converts a string into a floating-point number	`"5.7".toFloat()` → 5.7

The following program demonstrates the usage of different string methods listed in the previous tables. In this code, several sample strings are used and the corresponding string functions are applied to them to observe their behavior and output. The results are displayed through the Serial Monitor, allowing the user to understand how each method operates in practice.

Code 1.20 String

```
/*
  In this Code we will learn about Arduino
  "String" object and its different Methods.
*/

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

  String s = "Arduino Uno";

  Serial.println(s); // Arduino Uno
  Serial.println(s.length()); // 11
  Serial.println(s.charAt(0)); // A
  Serial.println(s.substring(0, 7)); // Arduino
  Serial.println(s.indexOf("Uno")); // 8

  s = s + " Board"; // s.concat(" Board");
  Serial.println(s); // Arduino Uno Board

  s.replace("Board", "Shield");
  Serial.println(s); // Arduino Uno Shield

  s.toUpperCase();
  Serial.println(s); // ARDUINO UNO SHIELD

  s.toLowerCase();
  Serial.println(s); // arduino uno shield
}

void loop() {

}
```

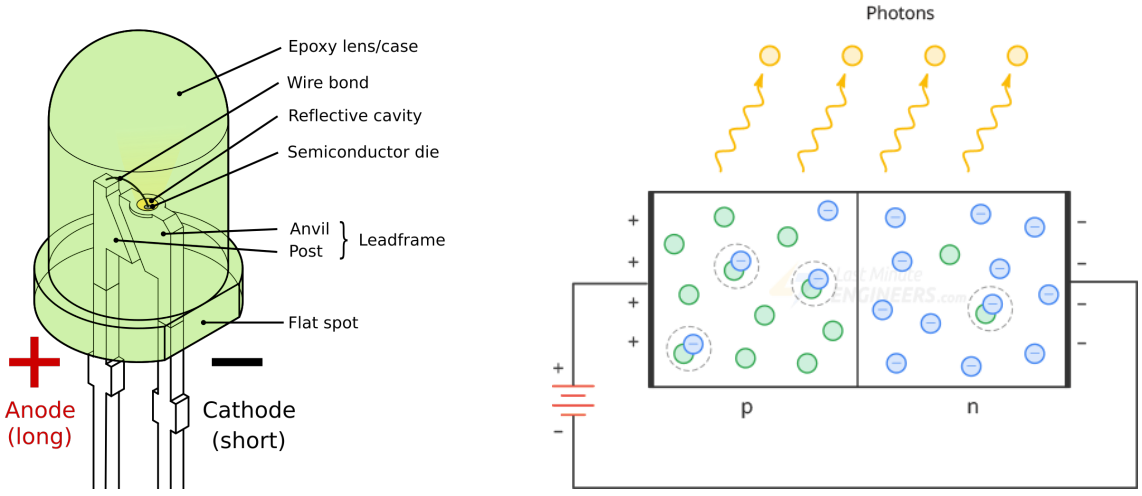
Output

```
Arduino Uno
11
A
Arduino
8
Arduino Uno Board
Arduino Uno Shield
ARDUINO UNO SHIELD
arduino uno shield
```

Chapter

2 Light Emitting Diode

A Light Emitting Diode (LED) is a semiconductor device that emits light when forward biased and electric current flows through it. Unlike traditional bulbs, LEDs consume very low power, generate minimal heat, and have a long operational lifespan. In electronic circuits, LEDs are commonly used as visual indicators for power status, signal activity, and debugging purposes. LEDs are so common, they come in a huge variety of shapes, sizes and colors. The most common LED configuration are the standard through hole LEDs with two legs. Like an ordinary diode, the LED operates only in forward bias condition. When the LED is forward biased, the free electrons cross the PN junction and recombine with holes. Since these electrons fall from a higher to a lower energy level, they radiate energy in the form of photons (light). In ordinary diodes, this energy is radiated as heat while in an LED, energy is radiated as light. This effect is called Electroluminescence.



(a) Internal Structure (b) Working Principle

Figure 2.1: LED Construction and Methodology

Digital IO

Digital Input/Output (I/O) pins of the Arduino UNO allow the microcontroller to interact with external electronic components by reading or generating digital signals. The Arduino UNO is equipped with 20 digital pins (D0-D13 & A0-A5), each of which can be configured either as an input or an output using software. A digital signal represents only two logical states: HIGH (typically 5V) and LOW (0V), making it suitable for interfacing with devices such as LEDs, push buttons, relays, logic circuits, and digital sensors. By setting a pin as OUTPUT, the Arduino can supply or sink current to control external loads, while configuring a pin as INPUT enables it to detect voltage levels applied from external sources. Some digital pins also support special functions such as PWM output, external interrupts, and serial communication, increasing the flexibility of the board. This simple yet powerful digital I/O capability forms the foundation of most embedded control and automation applications using Arduino UNO. Figure 2.2 shows the pin configuration of the Arduino Uno board, indicating the digital I/O pins, analog input pins, power pins, and communication interfaces.

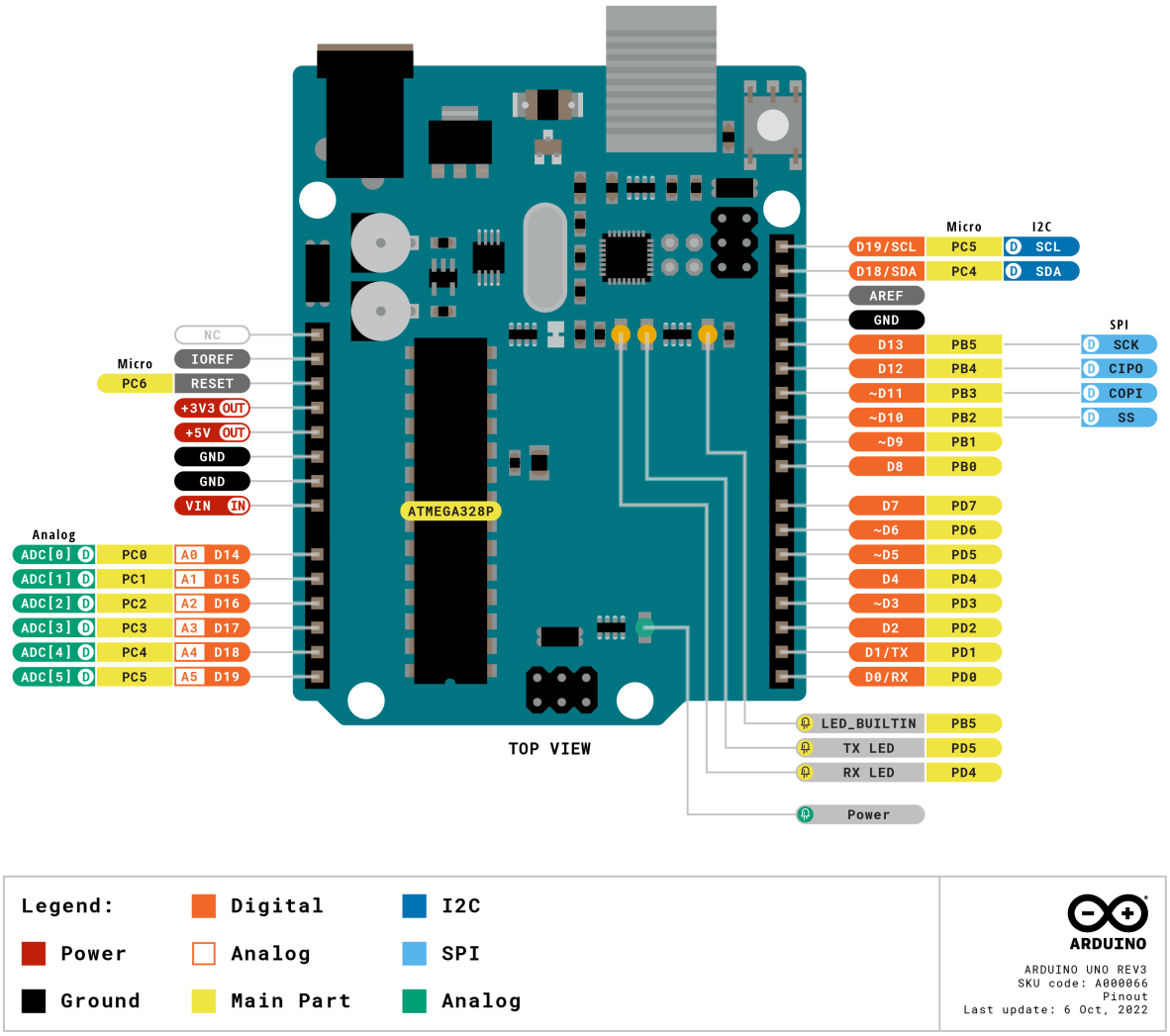


Figure 2.2: Arduino UNO Pinouts

Important Functions

For working with the digital I/O pins of the Arduino Uno, we will use some built-in functions. These functions are used in almost every digital input/output application and are essential for setting up the pins and managing their operation in Arduino-based projects. Three basic functions are commonly used in the program:

- `pinMode()`
- `digitalWrite()`
- `delay()`

Table 2.1: Digital IO Functions

Function	Purpose
`pinMode(pin, mode)`	Configures the specified digital pin to behave as set mode (Input or Output)
`digitalWrite(pin, value)`	Sets the output pin to logic value (Logic High – 5V or Logic Low – 0V)
`delay(ms)`	Introduces a blocking delay for the specified time in milliseconds (ms)

pinMode()

✍ `pinMode(pin, mode)`: Configures the operating mode of a digital pin.

- ☞ `OUTPUT / 1` : Configures the pin to drive logic levels (0V or 5V)
- ☞ `INPUT / 0` : Configures the pin to read external logic levels

digitalWrite()

✍ `digitalWrite(pin, value)`: Controls the logic level of a digital output pin.

- ☞ `HIGH / 1` : Outputs 5V, turning the LED ON
- ☞ `LOW / 0` : Outputs 0V, turning the LED OFF

Built-in LED Blink

Most Arduino boards have an on-board LED that can be controlled by the user. On the Arduino Uno, it is connected to digital pin 13. The constant `LED_BUILTIN` is set to the correct LED pin, independent of the board being used. We are going to turn this built-in LED on for one second and then off for one second, repeatedly.

Table 2.2: Digital Output

Function	Purpose
`pinMode(pin, OUTPUT)`	Configures the specified Arduino pin as a digital output pin
`digitalWrite(pin, HIGH)`	Sets the output pin to logic HIGH (5V)
`digitalWrite(pin, LOW)`	Sets the output pin to logic LOW (0V)
`delay(1000)`	Blocking delay for 1000ms = 1s

Code 2.1 Built-in LED Blink

```
/*
  Basic Blink
*/


// 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);       // pinMode(13, 1);

}


// the loop function runs over and over again forever
void loop() {

  // turn the LED on (HIGH is the voltage level)
  digitalWrite(LED_BUILTIN, HIGH);   // digitalWrite(13, 1);
  delay(1000);                       // wait for a second

  // turn the LED off (LOW is the voltage level)
  digitalWrite(LED_BUILTIN, LOW);    // digitalWrite(13, 0);
  delay(1000);                       // wait for a second

}
```

Current Limiting Resistor

An LED is a non-linear, "current-driven" device. Unlike a standard light bulb, it has very low internal resistance once it turns on. We need a resistor for three key reasons:

- **Prevention of Burnout:** If an LED is connected directly to a voltage source higher than its rating, it attempts to draw infinite current, resulting in immediate failure.
- **Voltage Drop:** The resistor absorbs the "excess" voltage from the power source that the LED does not use.
- **Current Limiting:** It restricts the flow of electrons to a safe level, ensuring the LED operates within its limits.

Different LED colors use different semiconductor materials. The energy bandgap of the material determines the color and the voltage required to turn it on (Forward Voltage or V_f).

Table 2.3: Different Color LEDs

LED Color	**Typical V_f**	**Typical Material**
Red	1.8V – 2.2V	AlGaAs / GaAsP / AlGaInP
Orange	2.0V – 2.2V	GaAsP / AlGaInP
Yellow	2.1V – 2.4V	GaAsP / AlGaInP / GaP
Green	2.0V – 3.2V	GaP / InGaN / AlGaInP
Blue	3.0V – 3.6V	InGaN / ZnSe / SiC
White	3.0V – 3.6V	InGaN / Yellow Phosphor

For standard LEDs:

- **Maximum Continuous Current (I_{max}):** 20mA
- **Recommended Current:** 10mA to 15mA

Note: Running an LED at the full 20mA provides maximum brightness but generates more heat. 10–15mA is the "sweet spot" for longevity.

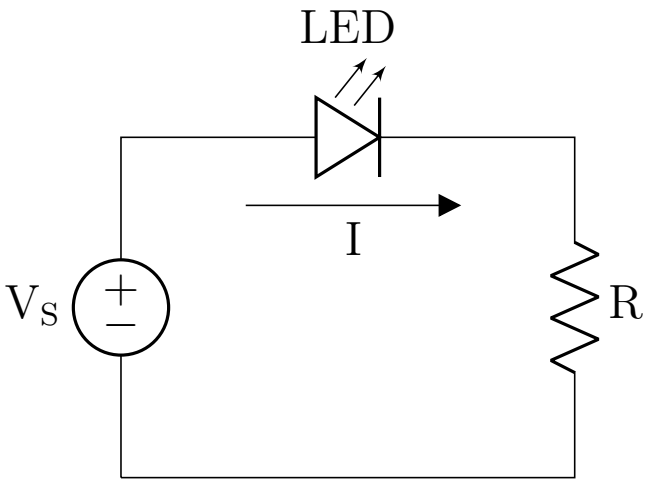


Figure 2.3: LED Circuit Connection

To calculate the required resistance, we use Ohm's Law:

$$R = \frac{V_R}{I_R} = \frac{V_S - V_{LED}}{I}$$

Where:

- V_S = Supply Voltage
- V_{LED} = LED Forward Voltage
- I = Desired Current

We are powering a **Red LED** using a **5V** supply at maximum safe brightness.

- Supply Voltage (V_S): 5V
- LED Forward Voltage (V_{LED}): 2V
- Target Current (I_{max}): 20mA

The resistor must drop the excess voltage:

$$V_R = V_S - V_{LED} = 5\text{V} - 2\text{V} = 3\text{V}$$

So, the minimum resitance required:

$$R_{min} = \frac{V_R}{I_{max}} = \frac{3\text{V}}{20\text{mA}} = 150\,\Omega$$

Based on the calculation, the absolute minimum resistance required to operate the LED at its maximum rating of 20mA is $150\,\Omega$. However, in practical circuit design, it is standard to round up to the next common resistor value to provide a safety margin and account for component tolerances. Therefore, a standard $220\,\Omega$ resistor is the recommended choice, ensuring the LED remains bright while operating safely below its absolute maximum limit.

External LED Blink

Now, we will control an external LED by integrating a current-limiting resistor into the circuit. The anode of the LED connects to a digital I/O pin, while the cathode connects in series to the resistor. The remaining terminal of the resistor is tied to ground (GND) as shown in Figure 2.4. Corresponding breadboard connection is shown in Figure 2.5 and codes are shown in next page.

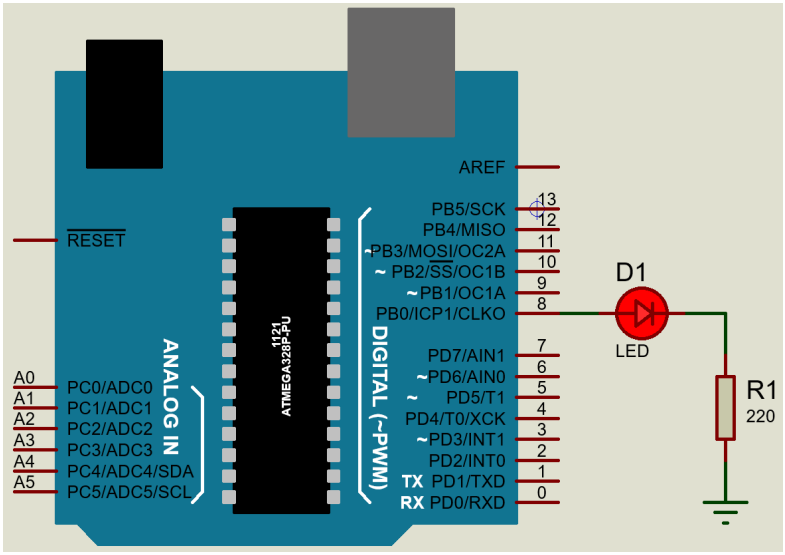


Figure 2.4: External LED Connection with Arduino

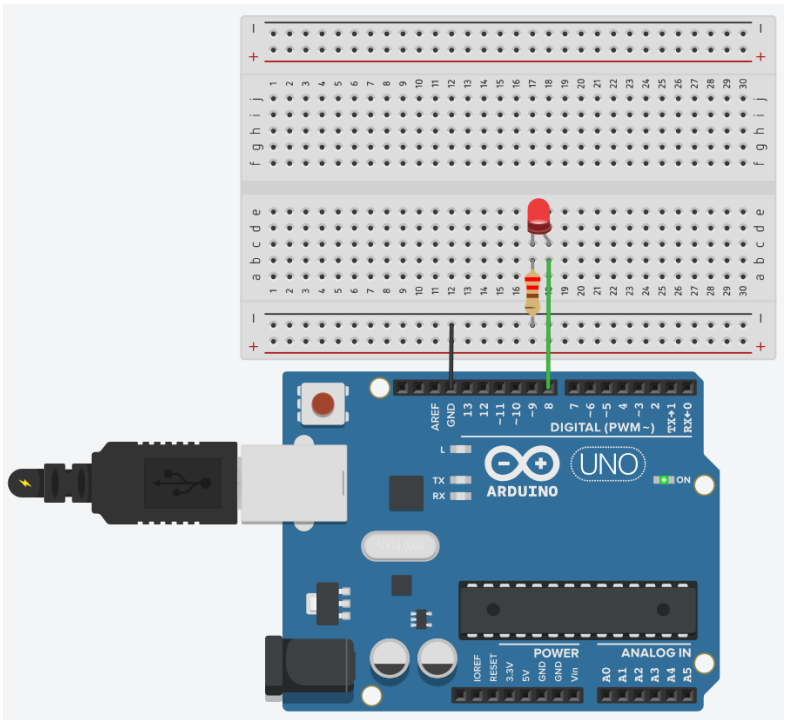


Figure 2.5: Breadboard Connection of Figure 2.4

Both of the following codes can be used for the setup shown in Figure 2.4 and Figure 2.5. They are essentially the same. Code 2.2 uses a direct approach, whereas Code 2.3 uses `#define` directive.

Code 2.2 External LED Blink

```
/* Turns an External LED on for one second, and then off for one second, repeatedly. */

void setup() {
  pinMode(8, OUTPUT);     // initialize digital LED pin as an output // pinMode(8,1);
}

void loop() {
  digitalWrite(8, HIGH);  // turn the LED on // digitalWrite(8,1);
  delay(1000);            // wait for a second
  digitalWrite(8, LOW);   // turn the LED off // digitalWrite(8,0);
  delay(1000);            // wait for a second
}
```

#define Directive

The `#define` directive is a preprocessor instruction used to define symbolic constants or simple macros. It is processed before the actual compilation of the program, meaning that the compiler replaces every occurrence of the defined name with the specified value throughout the code. This mechanism helps improve code readability and maintainability by allowing meaningful names to represent fixed values such as pin numbers, buffer sizes, or baud rates. For example, instead of repeatedly writing a specific pin number, a programmer can define a symbolic name using `#define` and use that name throughout the program. If the value needs to be changed later, it can be modified in a single place without altering the rest of the code. Because `#define` performs direct text substitution rather than creating a variable, it does not occupy memory and is widely used in Arduino programs for defining constants and simple macro expressions.

Code 2.3 #define Directive

```
/* Turns an External LED on for one second, then off for one second, repeatedly. */

// const int LED = 8; // Global Variable with Constant Qualifier

#define LED 8

void setup() {
  pinMode(LED, OUTPUT); // initialize digital LED pin as an output.
}

void loop() {
  digitalWrite(LED, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(LED, LOW);    // turn the LED off (LOW is the voltage level)
  delay(1000);               // wait for a second
}
```

User Interaction through Serial Monitor

The following code allows the user to input a number through the Serial Monitor, specifying how many times an LED should blink. This demonstrates the interaction between serial communication and digital I/O operations, where the Arduino reads the user input from the serial interface and controls the LED accordingly.

Code 2.4 'n' Times Blink

```
/*
   This code takes user input to determine how many times a LED would blink.
   It also shows a custom function for the purpose of blinking LED "n" times.
*/

# define LED 8

int n = 0; // no. of times LED would blink (User input)

void setup() {

  Serial.begin(9600); //Serial Monitor initialization
  pinMode(LED, OUTPUT); // Port Initialization

}

void loop() {

  // Taking USER Input
  Serial.print("How many times do you want LED to blink ? ");
  while (Serial.available() == 0) {}
  n = Serial.parseInt();
  Serial.println(n);

  // Calling custom user defined function for blinking LED
  blinkLED(n);

}

// Custom Function for Blinking LED
void blinkLED (int num) {

  // Code for blinking "n" times
  for (int i = 1; i <= num; i++)
  {
    digitalWrite(LED, HIGH);
    delay(500);
    digitalWrite(LED, LOW);
    delay(500);
  }

}
```

Controlling Multiple LEDs

Now, we will learn how to control multiple LEDs simultaneously. One green and one red LED are connected to two digital I/O pins as shown in Figure 2.6. Corresponding breadboard connection is shown in Figure 2.7. First, we would use PORT manipulation to control this setup as shown in Code 2.5. Then, we would use Bit manipulation as shown in Code 2.6.

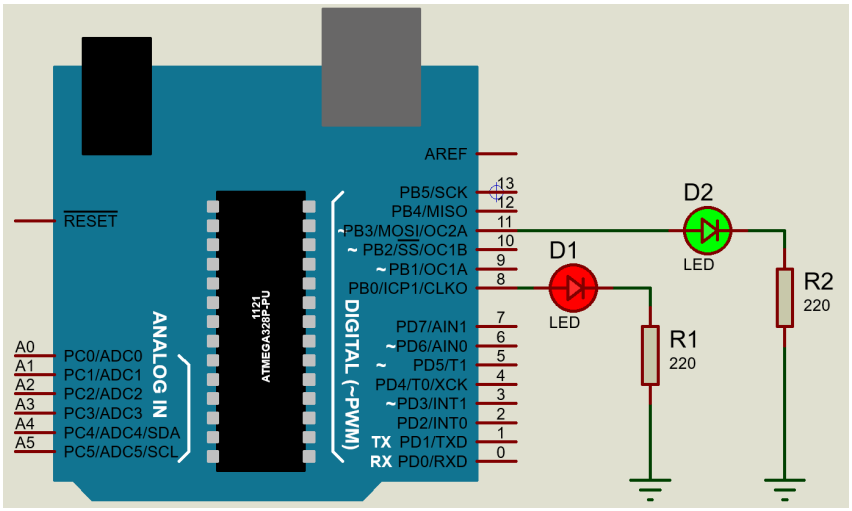


Figure 2.6: Red & Green LED Connection with Arduino

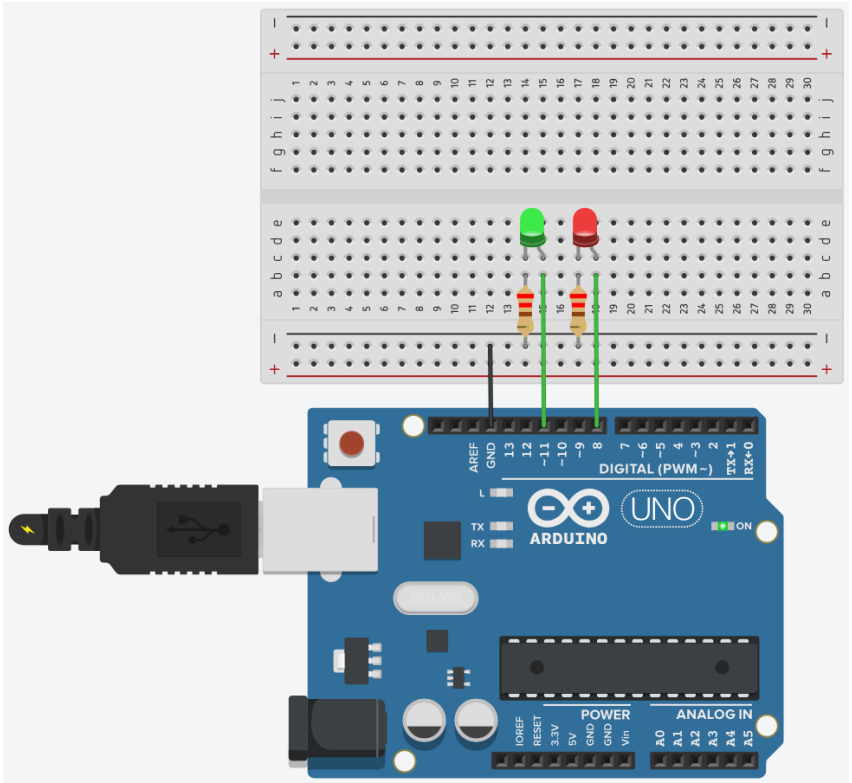


Figure 2.7: Breadboard Connection of Figure 2.6

Port Manipulation

Port manipulation refers to directly accessing and controlling the microcontroller's hardware registers to read from or write to multiple digital pins simultaneously, bypassing the standard Arduino functions. The Arduino UNO uses the ATmega328P microcontroller, where digital pins are grouped into ports. Each port is an 8-bit register, allowing all pins of a port to be controlled in a single instruction, which significantly increases execution speed and timing accuracy. By port manipulation, users can achieve faster signal toggling and precise control required in applications such as high-speed data communication, waveform generation, and real-time embedded systems. Although port manipulation improves performance and efficiency, it requires careful handling because it is hardware-dependent and less portable across different Arduino boards.

Table 2.4: Arduino UNO Ports

Port	**Arduino Pins**	**Bit Mapping**
PORTD	D7 – D0	PD7 – PD0
PORTB	D13 – D8	PB5 – PB0
PORTC	A5 – A0	PC5 – PC0

Table 2.5: Port Registers

Register	**Purpose**
DDRx	Data Direction Register
PORTx	Output register
PINx	Input register

Table 2.6: Configuration of Port Registers

Value	**DDRx**	**PORTx (OUTPUT mode)**
0	Pin configured as INPUT	Output LOW
1	Pin configured as OUTPUT	Output HIGH

Data Direction Register (DDR)

The Data Direction Register (DDR) is used to configure the function of each pin in a microcontroller port as either an input or an output. Each bit in the DDR corresponds to one physical pin, where a logic '1' sets the pin as an output and a logic '0' sets it as an input. By properly setting the DDR register (such as DDRB, DDRC, or DDRD in Arduino UNO), the user defines the direction of data flow before performing any read or write operation on the port pins.

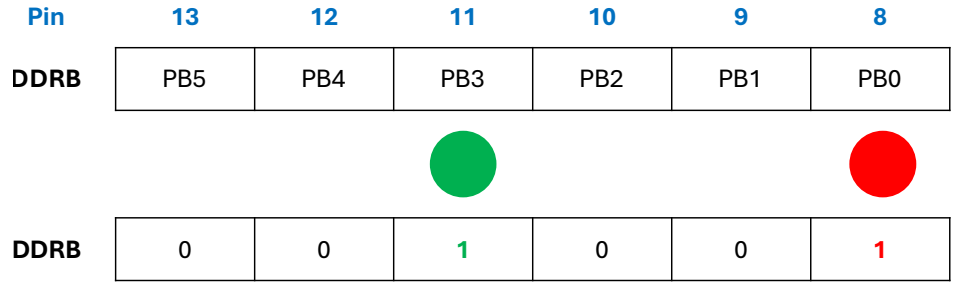


Figure 2.8: Data Directional Register (DDR)

PORT Register (PORT)

The PORT register is used to write data to pins configured as an output. Writing logic '1' to the corresponding PORT bit drives the pin HIGH (5V), and writing logic '0' drives it LOW (0V). The PORT register therefore controls the electrical behavior of the port pins during operation.

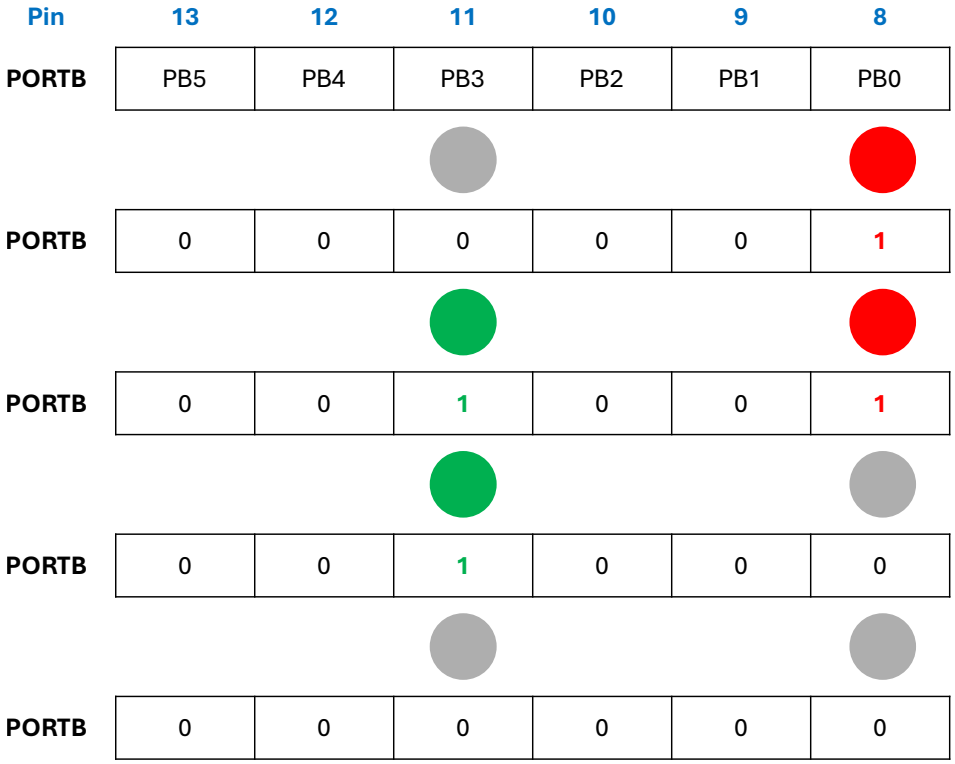


Figure 2.9: PORT Register (PORT)

In the following code, we will use Port Manipulation to control two LEDs (one Green LED & one Red LED) simultaneously for the setup shown in Figure 2.6 & Figure 2.7.

Here is the process flow we are trying to achieve –

- Green LED OFF, Red LED ON
- Green LED ON, Red LED ON (Both turned ON)
- Green LED ON, Red LED OFF
- Green LED OFF, Red LED OFF (Both turned OFF)

For this program, we would take help of Figure 2.8 and Figure 2.9 to set value of Data Direction Register (DDR) & PORT Register respectively.

Code 2.5 Port Manipulation

```
/*
  In this code we will learn about Port Manipulation.

  Here, we will control 2 LEDs simultaneously.
  One is RED LED connected to digital pin 8 (PB0) and
  other one is GREEEN LED connected to digital pin 11 (PB3).
*/


void setup() {
  DDRB = 0b001001; // setting pins D8(PB0) & D11(PB3) as Output Mode
}

void loop() {
  PORTB = 0b000001; // RED LED ON, GREEN LED OFF
  delay(1000);      // wait for a second

  PORTB = 0b001001; // BOTH LEDs ON
  delay(1000);      // wait for a second

  PORTB = 0b001000; // RED LED OFF, GREEN LED ON
  delay(1000);      // wait for a second

  PORTB = 0b000000; // BOTH LEDs OFF
  delay(1000);      // wait for a second
}
```

This technique allows for the simultaneous control of multiple I/O pins by bypassing the overhead of standard functions like `pinMode()` & `digitalWrite()`. By interacting directly with the microcontroller's registers, the code achieves significantly faster execution speeds and a smaller memory footprint, which is particularly useful for time-critical applications or projects with limited storage.

Bit Manipulation

In Arduino programming, bit manipulation can be easily performed using the built-in functions `bitRead()`, `bitWrite()`, `bitSet()`, and `bitClear()`, which allow direct control of individual bits within a variable or hardware register. The `bitRead()` function is used to read the value of a specific bit position, returning either 0 or 1. The `bitWrite()` function assigns a desired value to a particular bit without affecting other bits. Similarly, `bitSet()` forces a selected bit to logic 1, while `bitClear()` resets a selected bit to logic 0. These functions simplify low-level programming by providing readable and safe access to individual bits, making them especially useful for controlling digital I/O pins through registers, configuring peripheral settings, and handling status flags in embedded systems. For reference see Code 1.18 on Page 36.

In the following code, bit manipulation techniques are applied to simultaneously control two LEDs (one Green LED and one Red LED) for the setup shown in Figure 2.6 & Figure 2.7, maintaining the same execution sequence as the previous code.

- Green LED OFF, Red LED ON
- Green LED ON, Red LED ON (Both turned ON)
- Green LED ON, Red LED OFF
- Green LED OFF, Red LED OFF (Both turned OFF)

Code 2.6 Bit Manipulation

```
/*
  In this code we will learn about Bit Manipulation.

  Here, we will control 2 LEDs simultaneously.
  One is RED LED connected to digital pin 8 (PB0) and
  other one is GREEN LED connected to digital pin 11 (PB3).
*/


void setup() {
  bitSet(DDRB, 0); // Setting PB0 (D8) = 1 (Output Mode - RED LED)
  bitSet(DDRB, 3); // Setting PB3(D11) = 1 (Output Mode - GREEN LED)
}

void loop() {
  bitSet(PORTB, 0);      // RED led ON (GREEN LED OFF)
  delay(1000);           // wait for a second

  bitWrite(PORTB, 3, 1); // GREEN led ON (BOTH LEDs ON)
  delay(1000);           // wait for a second

  bitClear(PORTB, 0);    // RED led OFF (GREEN LED ON)
  delay(1000);           // wait for a second

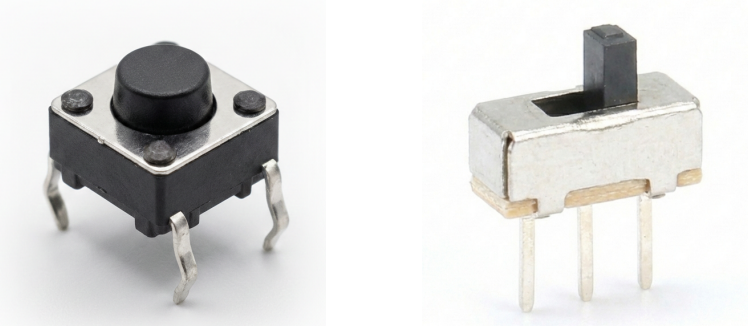
  bitWrite(PORTB, 3, 0); // GREEN led OFF (BOTH LEDs OFF)
  delay(1000);           // wait for a second
}
```

Chapter

3 SWITCH

A switch is a basic input device used in the Embedded System to control the flow of a program by providing user input. It works by opening or closing an electrical circuit, allowing the Arduino to detect a HIGH or LOW logic level on a digital input pin. By connecting a switch with a pull-up or pull-down resistor, false triggering due to noise can be avoided. Switches are widely used in Arduino projects for functions such as turning LEDs ON/OFF, menu selection, and user control interfaces. There are two types of actions pertinent to Switch. They are -

- Momentary Action (Example - Push Button)
- Latched Action (Example - Slide Switch)



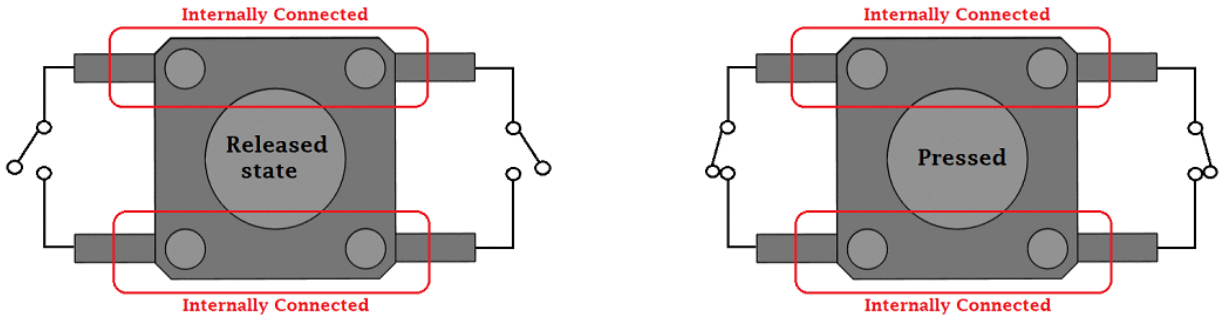
(a) Push Button (b) Slide Switch

Figure 3.1: Different Types of Switches

A momentary switch changes state only while it is being pressed and returns to its original position when released, making it suitable for temporary actions like doorbells or microcontroller inputs. In contrast, a latched (maintained) switch stays in its new state after being pressed or toggled and remains that way until it is switched again, which is why it is commonly used for power and mode selection.

Push Button

A 4-pin push button typically has a simple internal structure consisting of two pairs of pins that are internally connected in a cross configuration. When the button is not pressed, the circuit remains open, and no current flows between the pin pairs. Pressing the button brings the internal contacts together, connecting the pin pairs and allowing current to pass. The extra pins provide mechanical stability when the button is mounted on a breadboard and can also allow the button to be wired in multiple orientations without changing its functionality. This design makes the 4-pin push button reliable and easy to use in digital circuits for input detection.



(a) Released State (b) Pressed State

Figure 3.2: Push Button Structure

Slide Switch

A 3-pin slide switch has an internal structure with a movable contact (wiper) that connects the middle pin to one of the two outer pins, depending on the switch position. In one position, the middle pin is connected to the first outer pin, completing that circuit; in the other position, it connects to the second outer pin. This allows the switch to select between two paths or states. The third pin acts as a common terminal, while the two outer pins serve as selectable outputs. This simple design makes the 3-pin slide switch useful for controlling small electronic circuits or selecting modes in devices.

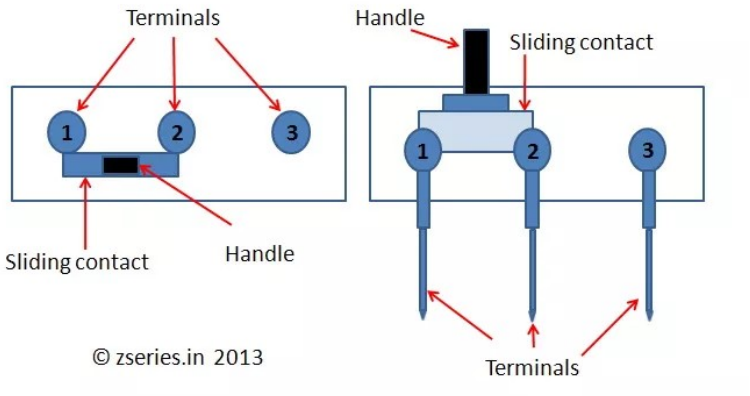


Figure 3.3: Slide Switch Structure

Floating Point Issue

In digital interfaces, an input pin is considered "floating" when it is not electrically connected to a voltage source (V_{cc}) or ground (GND). For example, when a push button is connected directly to an Arduino digital input pin, the pin is said to be in a **floating state** when the button is not pressed. A floating input does not have a fixed voltage level (neither 0V nor 5V), so it can randomly read HIGH or LOW due to electrical noise, leakage currents, or electromagnetic interference. As a result, the Arduino may detect false button presses even when the button is not touched. This behavior is unreliable and must be avoided in digital input circuits. To prevent floating, a resistor is used to force the input pin to a known voltage level when the button is not pressed. This resistor can be connected either to ground or to the supply voltage. The resistor ensures that the input pin always has a defined logic level, making the button reading stable and predictable.

Pull-Down Configuration

In the pull-down configuration, the resistor is connected between the input pin and ground (0V), and the push button connects the pin to 5V. When the button is not pressed, the resistor pulls the pin to LOW, so the Arduino reads logic 0. When the button is pressed, the pin is directly connected to 5V and the Arduino reads logic 1. This method works correctly, but it requires an external resistor, which increases component count and wiring complexity.

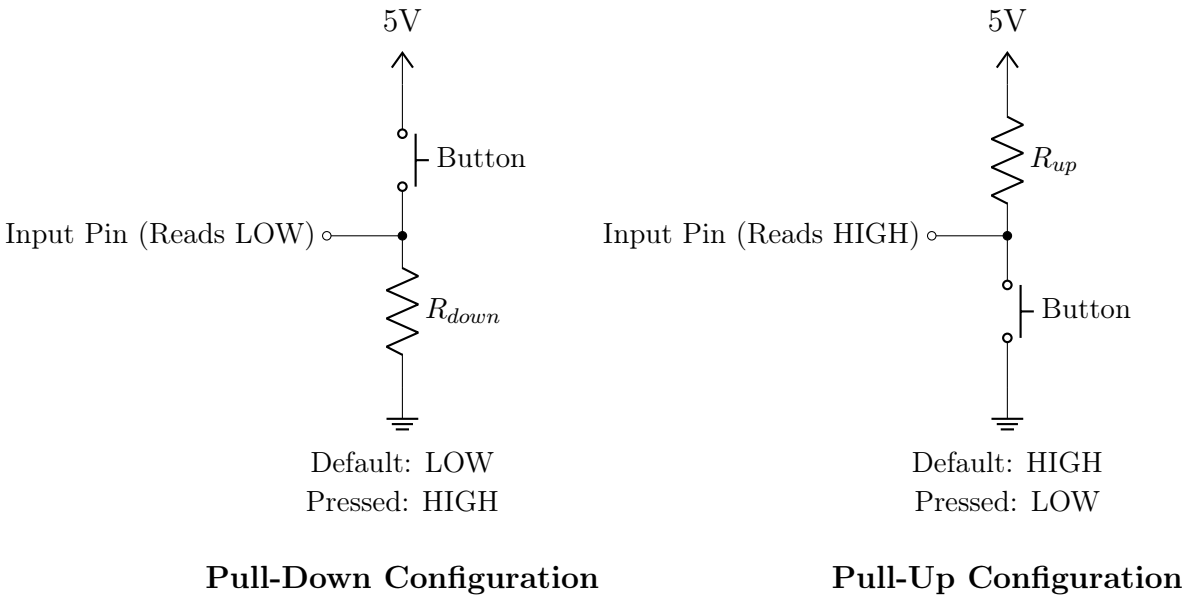


Figure 3.4: Pull-Down vs Pull-Up Configuration

Pull-Up Configuration

In the pull-up configuration, the resistor is connected between the input pin and 5V, and the push button connects the pin to ground. When the button is not pressed, the resistor pulls the pin to HIGH, so the Arduino reads logic 1. When the button is pressed, the pin is connected to ground and the Arduino reads logic 0. Arduino provides built-in internal pull-up resistors, so no external resistor is needed. This reduces hardware components and simplifies circuit design.

Important Functions

We would use some built-in functions to control the Digital Input. There are mainly two important functions related to Digital Input operation. Let's know the details of these two important functions.

pinMode()

☞ `pinMode(pin, mode)`: Configures the operating mode of the specified digital pin.

⇛ `OUTPUT`: Configures the pin to drive logic levels (0V or 5V)

⇛ `INPUT`: Configures the pin to read external logic levels

⇛ `INPUT_PULLUP`: Configures the pin as input with internal pull-up resistor enabled

digitalRead()

☞ `digitalRead(pin)`: Reads the value from a specified digital pin, either HIGH or LOW.

A Push Button

Now, first we will study the operation of a Push-Button. Now Push-Button can be used in 3 configurations. They are -

- External Pull-Down
- External Pull-Up
- Internal Pull-Up

Each configuration determines how the input pin behaves when the button is pressed or released, ensuring a stable and well-defined digital signal. The external pull-down configuration keeps the input at a LOW state when the button is not pressed, while the external pull-up configuration maintains a HIGH state by default. The internal pull-up configuration, on the other hand, uses the microcontroller's built-in resistor, reducing the need for additional external components. Now, we will examine these three common configurations in detail, along with their circuit diagrams and corresponding code. We will try to control an external LED using Push-button.

External Pull-Down

First, we will study the pull-down configuration of a push-button using an external resistor. The circuit schematic is shown in Figure 3.5, and the corresponding breadboard layout is shown in Figure 3.6. We will use Code 3.1 for this setup.

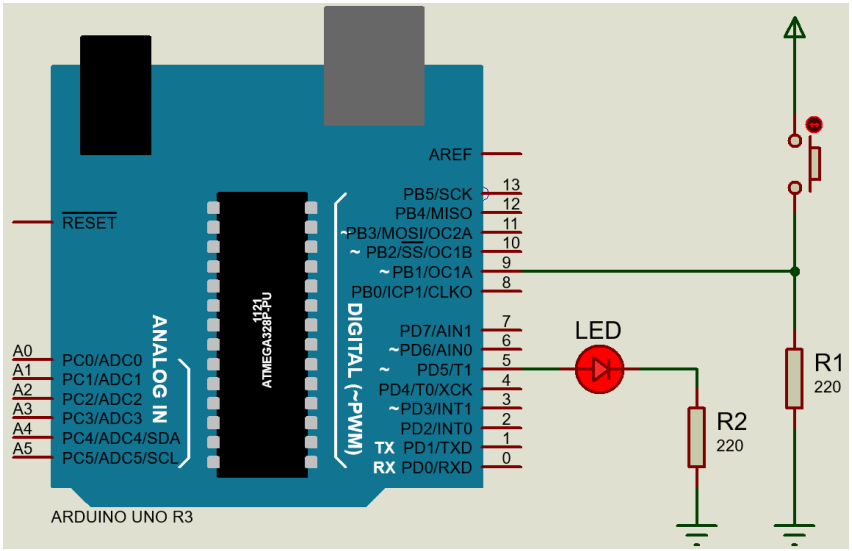


Figure 3.5: External Pull-Down

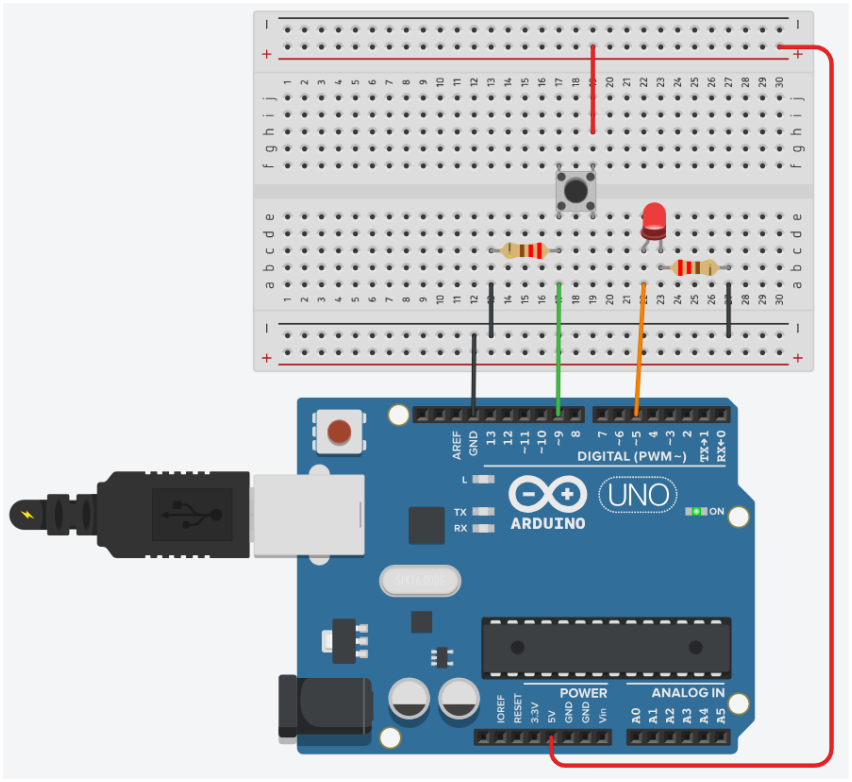


Figure 3.6: Breadboard Connection of Figure 3.5

In this program, the push-button pin is set as an input and the LED pin as an output, and serial communication is started so the button state can be observed on the Serial Monitor. During execution, the board continuously reads the voltage level at the push-button pin. Since an external pull-down resistor is used, the default state of the button is LOW (0V) when it is not pressed. When the button is pressed, the pin receives HIGH (5V). The program prints this 1 (HIGH) or 0 (LOW) value to the Serial Monitor, allowing us to see whether the button is pressed or not. At the same time, the same value is sent to the LED pin, so the LED turns ON when the button is pressed (HIGH) and turns OFF when it is released (LOW).

Code 3.1 External Pull-Down

```
/*  This code controls a LED through the use of a Push-Button.

    The Push-Button is connected to digital pin 9
    and the LED is connected to digital pin 5.

    If the Push-Button is pressed, the LED is turned ON.
    Otherwise, when the button is not pressed, the LED is turned OFF.

    This type of connection is known as External Pull-Down.
    Here, we are using an "External" resistance which "Pulls Down"
    voltage to Zero (0 --> LOW) Voltage when the button is not pressed.
*/


// Setting Pin Connection for Push-Button & LED
# define PB 9
# define LED 5


void setup() {

  // Setting Pin Mode for Push-Button & LED
  pinMode(PB, INPUT);
  pinMode(LED, OUTPUT);

  // Initializing Serial Monitor
  Serial.begin(9600);

}

void loop() {

  // Reading the State of the Push-Button
  int buttonState = digitalRead(PB); // Push-Button: 1 --> HIGH(5V) or 0 --> LOW(0V)

  // Printing the State of the Push-Button
  Serial.println(buttonState);

  // Writing the State of the LED
  digitalWrite(LED, buttonState); // LED: HIGH --> ON & LOW --> OFF

}
```

External Pull-Up

Next, we will study the pull-up configuration of a push-button using an external resistor. The circuit schematic is shown in Figure 3.7, and the corresponding breadboard layout is shown in Figure 3.8. We will use Code 3.2 for this setup.

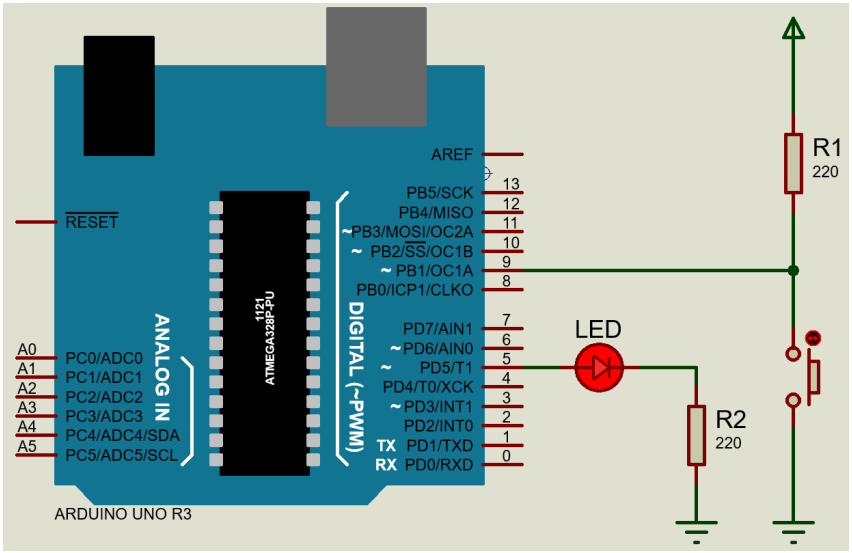


Figure 3.7: External Pull-Up

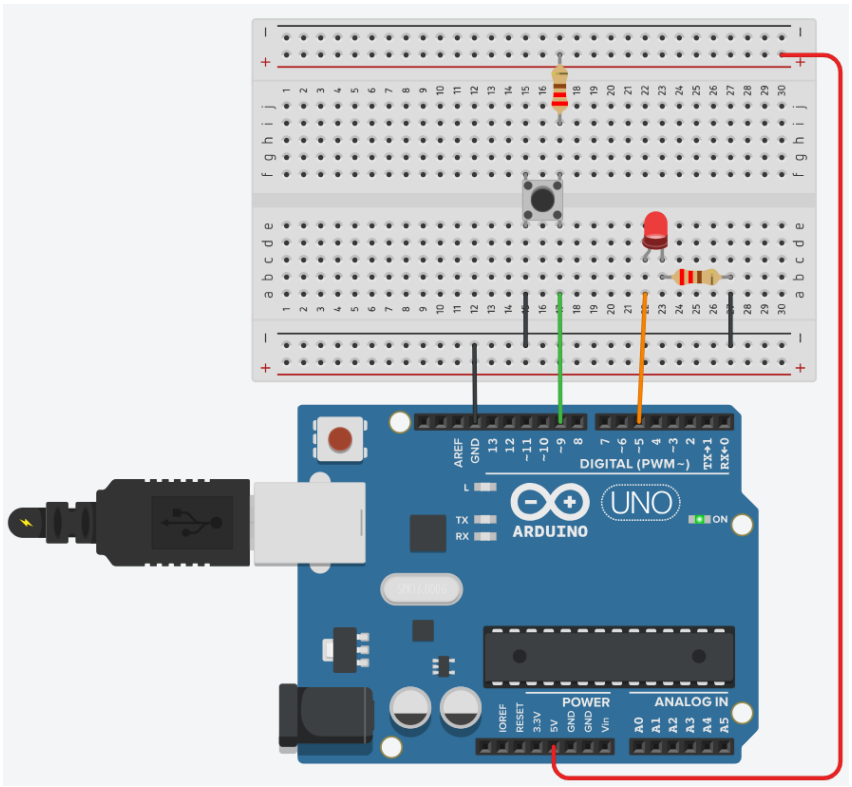


Figure 3.8: Breadboard Connection of Figure 3.7

In this setup, the button pin at HIGH (5V) when the button is not pressed, making HIGH the default state. When the button is pressed, the pin is connected to ground, producing a LOW (0V) signal. The program continuously reads the button state and prints its value (1 = HIGH or 0 = LOW) to the Serial Monitor for observation. Instead of directly writing the read value to the LED as in the previous example, an `if-else` statement is used to control the LED explicitly. Although the control structure is different, the overall process remains the same: the button state is checked, and based on that state, the LED is driven accordingly. By modifying the conditions inside the `if-else` statement, the default LED behavior can also be changed.

Code 3.2 External Pull-Up

```
/*  This code controls a LED through the use of a Push-Button.

    The Push-Button is connected to digital pin 9
    and the LED is connected to digital pin 5.

    If the Push-Button is pressed, the LED is turned OFF.
    Otherwise, when the button is not pressed, the LED is turned ON.

    This type of connection is known as External Pull-Up.
    Here, we are using an "External" resistance which "Pulls Up"
    voltage to 5V(HIGH) when the button is not pressed.
*/


// Setting Pin Connection for Push-Button & LED
# define PB 9
# define LED 5


void setup() {

  // Setting Pin Mode for Push-Button & LED
  pinMode(PB, INPUT);
  pinMode(LED, OUTPUT);

  // Initializing Serial Monitor
  Serial.begin(9600);
}

void loop() {

  // Reading the State of the Push-Button
  int buttonState = digitalRead(PB); // Push-Button: 1 --> HIGH(5V) or 0 --> LOW(0V)

  Serial.println(buttonState); // Printing the State of the Push-Button

  // Writing the State of the LED
  if (buttonState == HIGH)
    digitalWrite(LED, HIGH);
  else
    digitalWrite(LED, LOW);
}
```

Internal Pull-Up

Now, we will study the internal pull-up configuration of a push-button using the Arduino's built-in pull-up resistor. The circuit schematic is shown in Figure 3.9, and the corresponding breadboard layout is shown in Figure 3.10. We will use Code 3.3 for this configuration.

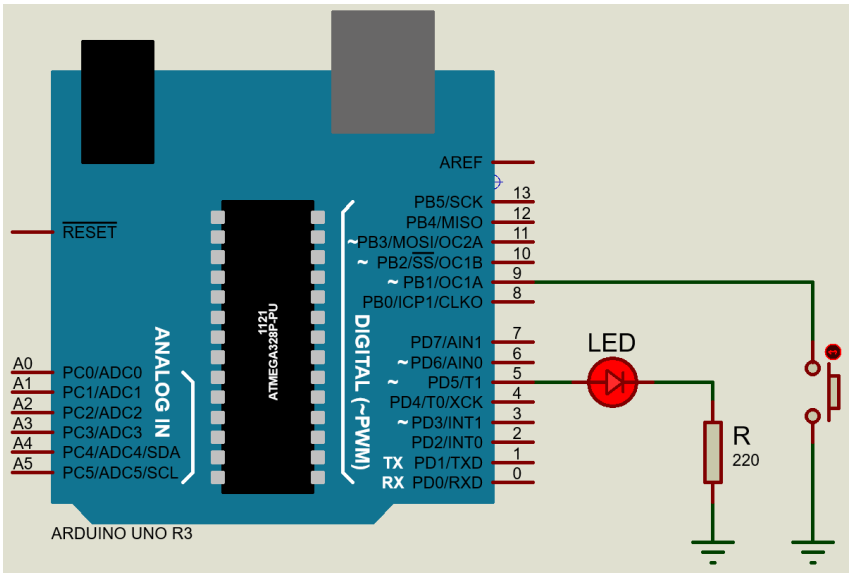


Figure 3.9: Internal Pull-Up

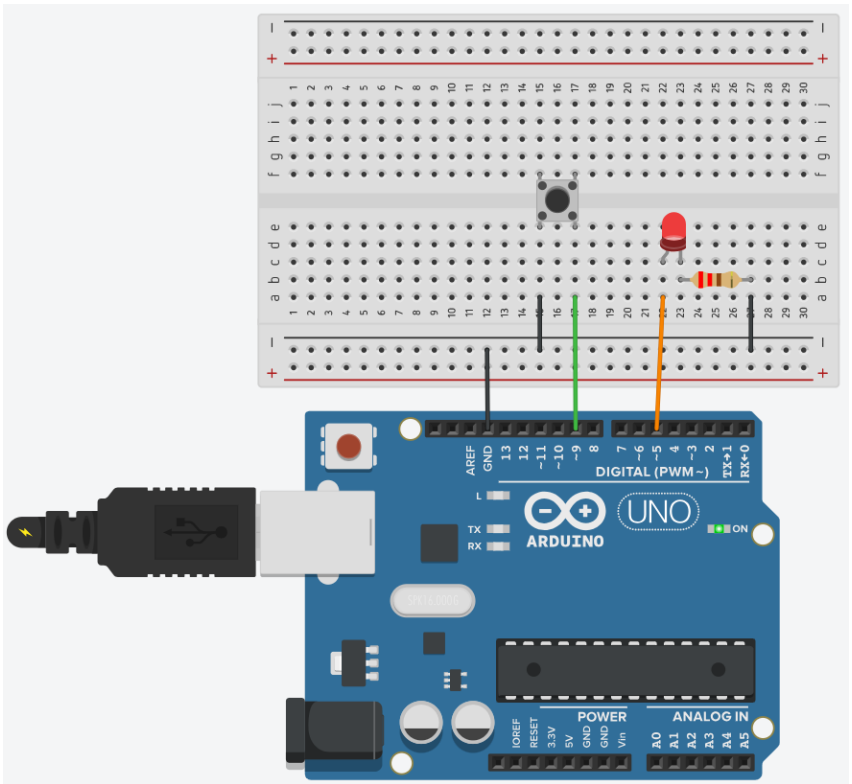


Figure 3.10: Breadboard Connection of Figure 3.9

This program demonstrates the use of an internal pull-up resistor. The only change compared to the external resistor configuration is in the `pinMode()` setting, where the button pin is defined as `INPUT_PULLUP`. This activates the built-in internal resistor inside the Arduino, eliminating the need for an external resistor. With this configuration, the default state of the button pin is HIGH (5V) when the button is not pressed, and it becomes LOW (0V) when the button is pressed because the pin is connected to ground. The program continuously reads and prints the button state to the Serial Monitor. Since the pressed state produces LOW, the LED will normally turn OFF when the button is pressed if the state is written directly. However, by applying the logical NOT operator (!) to invert the button state, the logic can be reversed so that pressing the button turns the LED ON, which is more intuitive than the default behavior.

Code 3.3 Internal Pull-Up

```
/*  This code controls a LED through the use of a Push-Button.

    The Push-Button is connected to digital pin 9
    and the LED is connected to digital pin 5.

    If the Push-Button is pressed, the LED is turned OFF.
    Otherwise, when the button is not pressed, the LED is turned ON.

    This type of connection is known as Internal Pull-Up.
    Here, we are NOT using any External resistance.
    Instead we will use "Internal" resistance inside Arduino.
    This internal resistance "Pulls Up" voltage to
    5V(HIGH) when the button is not pressed.
*/

// Setting Pin Connection for Push-Button & LED
# define PB 9
# define LED 5

void setup() {

  // Setting Pin Mode for Push-Button & LED
  pinMode(PB, INPUT_PULLUP); // -- PULLUP -- //
  pinMode(LED, OUTPUT);

  // Initializing Serial Monitor
  Serial.begin(9600);
}

void loop() {

  // Reading the State of the Push-Button
  int buttonState = digitalRead(PB); // Push-Button: 1 --> HIGH(5V) or 0 --> LOW(0V)

  Serial.println(buttonState); // Printing the State of the Push-Button

  // Writing the State of the LED
  digitalWrite(LED, buttonState); // Use (!) to invert State
}
```

B Slide Switch

Finally, we will study the operation of a slide switch. As Slide-Switch does not have any floating states, we can control an LED using only slide-switch itself without needing any external resistances. The circuit schematic is shown in Figure 3.11, and the corresponding breadboard layout is shown in Figure 3.12. We will use Code 3.4 for this configuration.

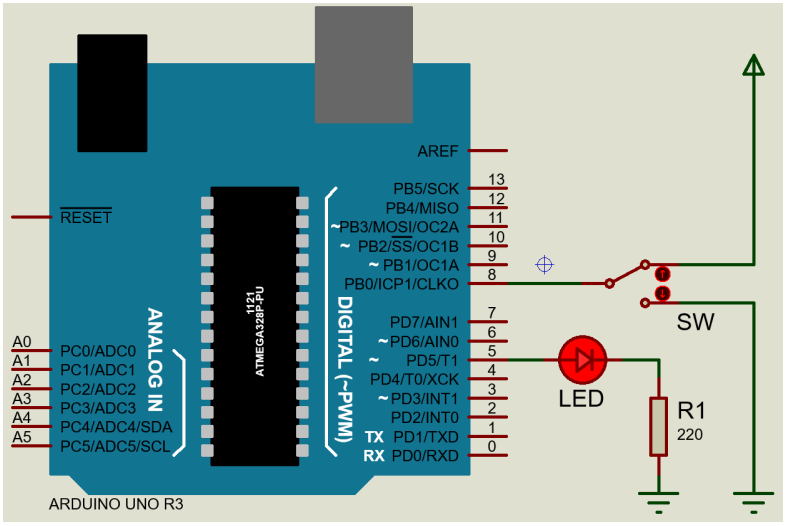


Figure 3.11: Slide Switch

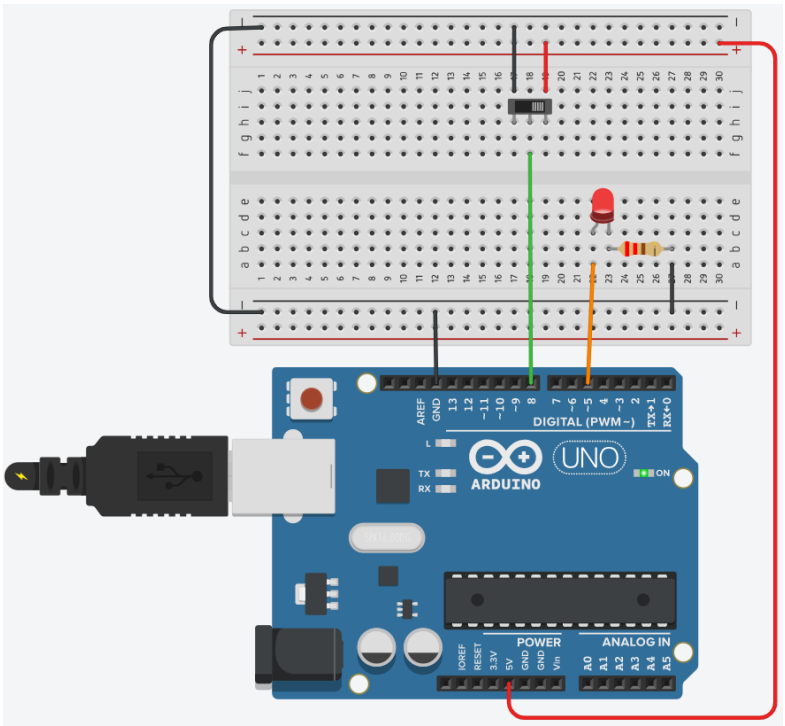


Figure 3.12: Breadboard Connection of Figure 3.11

In this program, the slide-switch pin is configured as an input and the LED pin is configured as an output. During execution, the board continuously reads the digital state (HIGH or LOW) of the slide-switch. This same digital value is then written directly to the LED. As a result, the LED state follows the slide-switch state exactly — when the switch outputs HIGH, the LED turns ON, and when it outputs LOW, the LED turns OFF.

Code 3.4 Slide Switch

```
/*  This code controls a LED through the use of a Slide-Switch.

    The Slide Switch is connected to digital pin 10
    and the LED is connected to digital pin 7.

    If the switch is slided to 5V(HIGH), the LED is turned ON.
    Otherwise, when the switch is slided to 0V(LOW), the LED is turned OFF.
*/


// Setting Pin Connections for Slide-Switch & LED
# define SW 8
# define LED 5


void setup() {

  // Setting Pin Mode for Slide-Switch & LED
  pinMode(SW, INPUT);
  pinMode(LED, OUTPUT);
}

void loop() {

  // Input Switch to Output LED
  digitalWrite(LED, digitalRead(SW));
}
```

Logic Gate Implementation

Now, we are going to implement different logic gates using couple of slide switches and an LED. Here, two slide switches are used as digital inputs to realize basic logic gate operations such as AND, OR, NAND, NOR, XOR, and XNOR using the Arduino. Each slide switch acts as one input, labeled Input A and Input B, and each can be set to either HIGH or LOW, representing logic '1' and logic '0', respectively. An LED connected to a digital output pin is used to indicate the result of the logic operation. By changing the positions of the two switches, all four possible input combinations (00, 01, 10, and 11) can be applied and the corresponding output can be observed directly on the LED. For each logic gate, only the program code needs to be changed while the same hardware connections are retained, allowing students to clearly compare the truth tables and behavior of different logic functions through hands-on observation.

The circuit schematic is shown in Figure 3.13, and the corresponding breadboard layout is shown in Figure 3.14. Using this setup, Code 3.5 implements the AND gate.

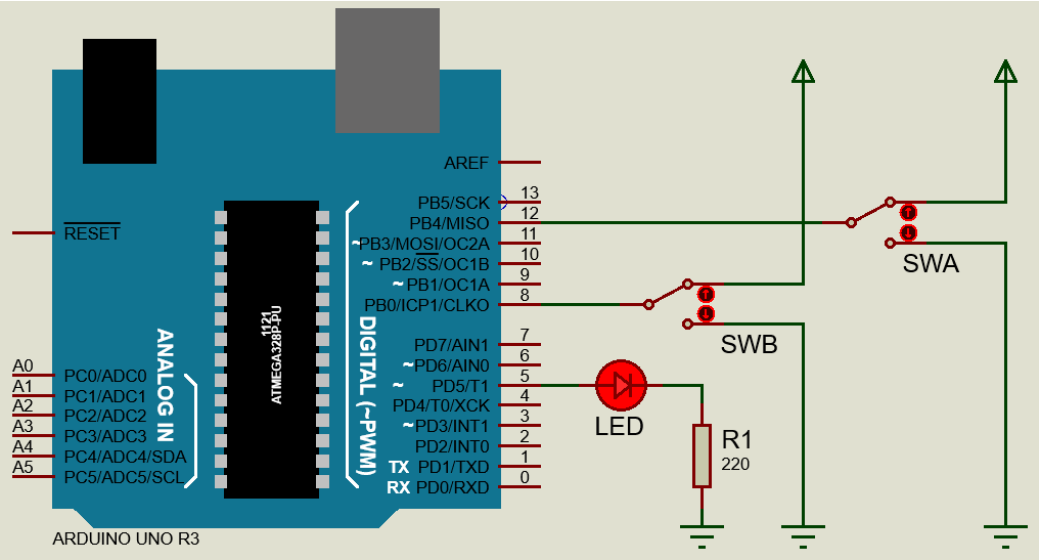


Figure 3.13: Logic Gate

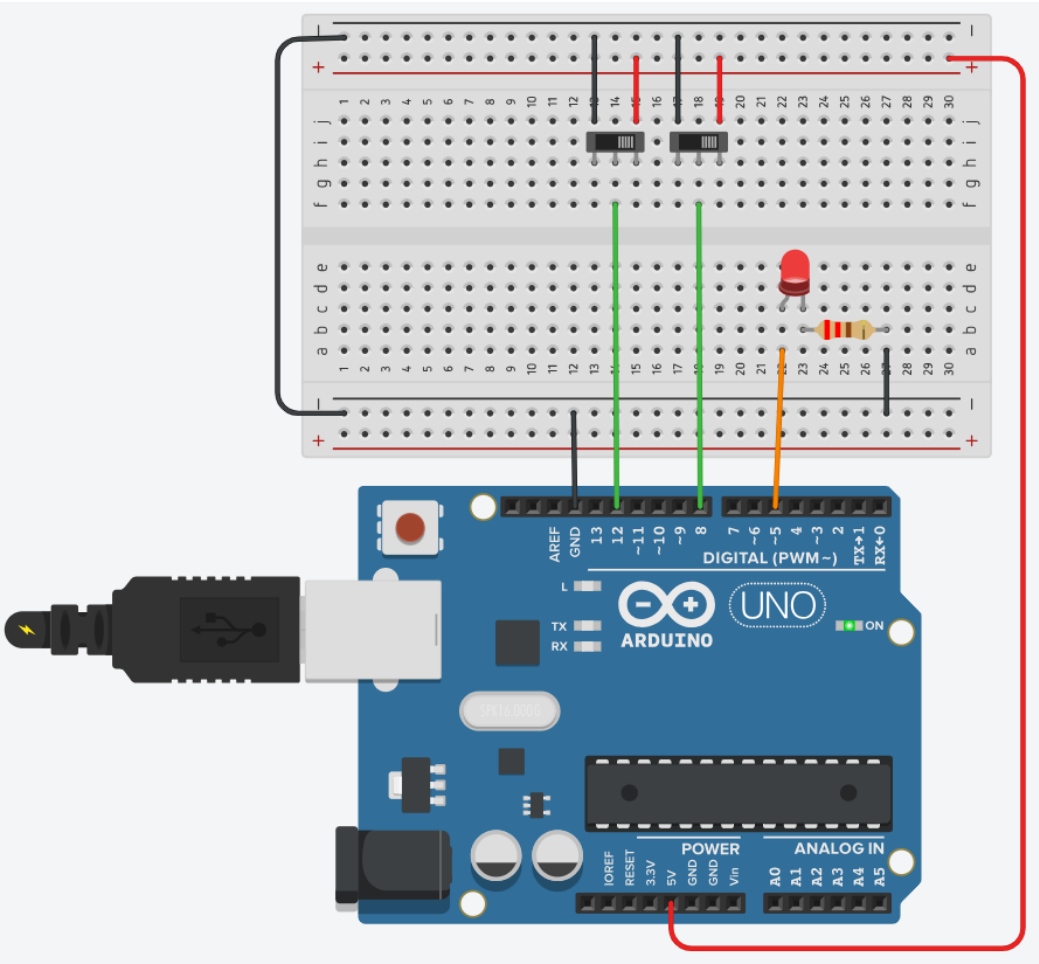


Figure 3.14: Breadboard Connection of Figure 3.13

Code 3.5 Logic Gate

```
/*  This code implements Logic Gate using Slide-Switches.

    Here we would implement a 2-input AND gate.
    Two Slide-Switches are used for inputs and
    a single LED is used to show output.
*/


// Setting Pin Connections for Slide-Switch & LED
# define SWA 12
# define SWB 8
# define LED 5


void setup() {

  // Setting Pin Mode for Slide-Switch & LED
  pinMode(SWA, INPUT);
  pinMode(SWB, INPUT);
  pinMode(LED, OUTPUT);

}

void loop() {

  // Inputs A & B Switch
  int A = digitalRead(SWA);
  int B = digitalRead(SWB);

  // ** Change the expression for other Logic Gates ** //
  digitalWrite(LED, A & B); // AND (&) Gate

}
```

The above code implements an AND gate. Other logic gates can be implemented easily by simply modifying a single line of the program, replacing the logical expression with the appropriate operator. The required expressions for other logic gates are summarized in the following table.

Table 3.1: Logic Gates

Logic Gate	Expression	Logic Gate	Expression
AND	A & B	NAND	! (A & B)
OR	A | B	NOR	! (A | B)
XOR	A ∧ B	XNOR	! (A ∧ B)

Chapter

4 Liquid Crystal Display

A Liquid Crystal Display (LCD) is a flat-panel display technology widely used in electronic systems to present alphanumeric characters, symbols, and graphics with low power consumption and compact size. The operating physics of an LCD is based on liquid crystal materials that exhibit properties between liquids and solids; these molecules can rotate their orientation when an electric field is applied, thereby controlling the polarization of light passing through them. An LCD itself does not emit light; instead, it modulates ambient light or light from a backlight using polarized filters and liquid crystal layers. In a typical character LCD (such as 16×2), the structure consists of a backlight source (LED), a rear polarizer, a liquid crystal layer sandwiched between transparent electrodes, a front polarizer, and patterned segments arranged as characters or pixels. When voltage is applied across specific segments, the liquid crystal alignment changes, blocking or allowing light to pass and thus forming visible characters. The commonly used pinout for an HD44780-compatible LCD includes power pins (VSS for ground and VDD for 5V), a contrast control pin (V0), control pins (RS for register select, R/W for read/write, and E for enable), data pins (D0–D7, often used in 4-bit or 8-bit mode), and backlight pins (A and K for LED anode and cathode).

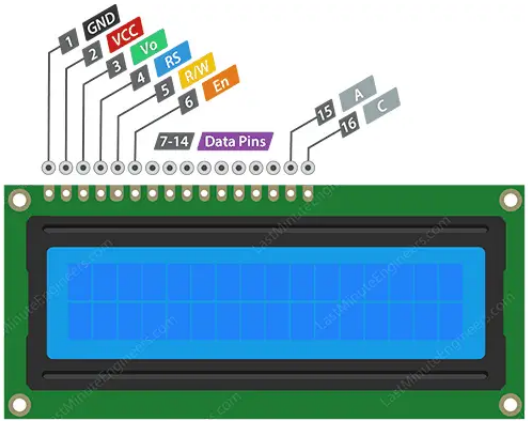


Figure 4.1: LCD Pinouts

An I2C LCD is a LCD module integrated with an I2C interface adapter, allowing the display to communicate with a microcontroller using only two data lines instead of many parallel pins. Internally, the LCD is usually a standard HD44780-compatible module, while the I2C adapter is built around an I/O expander IC (most commonly PCF8574 or PCF8574A). The working principle remains the same as a normal LCD: liquid crystal segments control light transmission based on applied voltage, but the control signals (RS, RW, E, and data lines) are generated indirectly through the I2C expander.

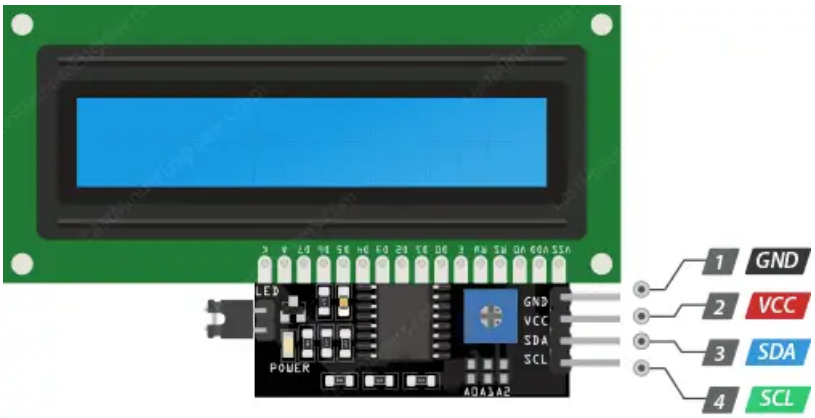


Figure 4.2: I2C LCD Pinouts

The structure consists of the LCD panel, backlight, and a small daughter board mounted on the back that contains the I2C expander, pull-up resistors for SDA and SCL, a contrast potentiometer, and address-selection jumpers. The typical pinout of an I2C LCD module is limited to four pins: VCC, GND, SDA (serial data), and SCL (serial clock), which connect directly to the microcontroller's I2C bus. The I2C address of the module is usually 0x27 or 0x3F, depending on the expander type and jumper configuration. They are widely used in embedded systems such as Arduino, digital measuring instruments, and control panels, especially where GPIO pins are limited. Their advantages include reduced wiring complexity, easier PCB routing, support for multiple devices on the same bus, and simpler code management, making I2C LCDs a practical and scalable solution for text-based user interfaces in embedded applications.

Table 4.1: I2C LCD Connection with Arduino

LCD	**Arduino**
VCC	5V
GND	GND
SDA	A4
SCL	A5

Finding Slave Address

Before using the I2C LCD, it is important to know its slave address, because different LCD modules may use different I2C expander chips and configurations. Therefore, a simple I2C scanner program is first uploaded to the Arduino to detect the connected device address automatically. Most commonly, LCD modules with the PCF8574 expander use address 0x27, while modules with PCF8574A use 0x3F, depending on the internal hardware wiring and address pins. Once the correct address is found using the scanner, the same address must be used in the LCD program; otherwise, the display will not respond even if all connections are correct.

To determine the slave address, the Arduino Uno scans a range of possible addresses commonly used by such devices. The microcontroller activates the I2C interface to communicate with devices on the bus. It then sequentially checks each address within the specified range by attempting to establish communication. If a device responds with an acknowledgment, that address is identified as the slave address.

Code 4.1 I2C Address

```
/*
   This code is used to find the slave address of 8 bit I/O Extender (PCF8574/PCF8574A)
   connected with the LCD which would be used for I2C communication with Arduino UNO.
*/


// Library for communicating with I2C / TWI devices
#include <Wire.h>

byte n; // Variable for finding device address

void setup() {

  Serial.begin (9600);
  Serial.println ("Finding I2C Devices ...");

  //Initiating "Wire" library and joining the I2C bus as a master
  Wire.begin();

  // Address is between 0x20 - 0x3F (32 - 63)
  for (n = 32; n <= 63; n++) {
    Wire.beginTransmission (n);//Starting transmission with I2C slave device

    //Ending Successful transmission
    if (Wire.endTransmission () == 0) {
      Serial.print ("Slave Address: 0x");
      Serial.println (n, HEX);
      break;
    }
  }
}

void loop() {

}
```

Basic LCD Printing

Basic LCD printing involves displaying text or numerical data on an LCD screen using simple commands provided by the I2C LCD library in Arduino. It begins with initializing the display, turning the backlight LED on, and sending data using printing functions. Understanding these fundamental steps helps users learn how information is shown on the LCD and how the display can be controlled effectively. The following section introduces the essential methods required for basic LCD operation and demonstrates a simple program to print `Hello World!` providing a clear starting point for LCD interfacing.

Table 4.2: Basic Methods

Method	Description	Default
.init()	Initializes the I2C LCD	-
.backlight()	Turns LCD backlight ON	-
.noBacklight()	Turns LCD backlight OFF	Yes
.print()	Prints Message on LCD	-

! If I2C scanner shows that address is 0x27, you must replace 0x3F with 0x27 in code.

Code 4.2 Hello World

```
/* This sketch prints "Hello World!" to the LCD Display. */

// include the library
#include <LiquidCrystal_I2C.h>

// Declaring a object
LiquidCrystal_I2C lcd(0x3F, 16, 2); //Slave Address, Column, Row

void setup() {

  lcd.init(); // Initialize the LCD

  lcd.backlight(); // Turns on the backlight of the LCD

  lcd.print("Hello World!"); // Print a message to the LCD

}

void loop() {

}
```

Common Built-in Methods

A variety of built-in functions are provided by the I2C LCD library to simplify the control and operation of LCD displays. Some commonly used methods are listed in the following two tables.

Table 4.3: Display & Cursor Methods

Method	Description	Default
.home()	Moves cursor to first row, first column	Yes
.clear()	Clears LCD screen	-
.display()	Turns LCD display ON	Yes
.noDisplay()	Turns LCD display OFF	-
.setCursor(col, row)	Sets cursor position	-
.cursor()	Shows cursor	-
.noCursor()	Hides cursor	Yes
.blink()	Enables blinking cursor	-
.noBlink()	Disables blinking cursor	Yes

Table 4.4: Text Direction & Scrolling Methods

Method	Description	Default
.leftToRight()	Sets text direction left to right	Yes
.rightToLeft()	Sets text direction right to left	-
.scrollDisplayLeft()	Scrolls display to the left	-
.scrollDisplayRight()	Scrolls display to the right	-
.autoscroll()	Automatically scrolls text	-
.noAutoscroll()	Disables auto scrolling	Yes

Controlling LCD Text Position

The `setCursor()` method is used to control the exact position where text appears on the LCD display. Since an LCD screen consists of multiple rows and columns, this function allows the user to specify the cursor location by defining the column and row numbers before printing any data. It helps organize information clearly by displaying different messages on different lines or at specific positions on the screen. This feature is especially useful in embedded system applications where structured output, such as sensor readings, menus, or status messages, needs to be presented in an orderly manner. By using the `setCursor()` function, users can efficiently manage screen space and improve the readability and functionality of LCD-based Arduino projects.

Code 4.3 Set Cursor

```
/*

  This sketch prints "Ahsanullah University" to the LCD Display.
  It prints "Ahsanullah" on Top Line and then prints "University" on Bottom Line.

*/


// include the library
#include <LiquidCrystal_I2C.h>


// Declaring a object
LiquidCrystal_I2C lcd(0x3F, 16, 2); //Slave Address, Column, Row


void setup() {

  lcd.init(); // Initialize the LCD

  lcd.backlight(); // Turns on the backlight of the LCD

  // 1st argument refers display block 3 (Column); 2nd argmunet refers TopLine (Row)
  lcd.setCursor(3, 0); // (Column Number, Row Number)

  // Print a message on the Topline of the LCD.
  lcd.print("Ahsanullah");

  // 1st argument refers display block 3 (Column); 2nd argmunet refers BottomLine (Row)
  lcd.setCursor(3, 1); // (Column Number, Row Number)

  // Print a message on the Topline of the LCD.
  lcd.print("University");
}


void loop() {

}
```

LCD Printing & Serial Printing

The `lcd.print()` function works very similarly to the `Serial.print()` function, as both provide flexible ways to display data in different formats. They allow users to print text, integers, and floating-point numbers with controlled formatting. For floating-point values, the second argument specifies the number of digits to display after the decimal point, while for integer values it determines the number system used for representation, such as decimal, hexadecimal, or binary. This formatting capability helps present data clearly and meaningfully on the LCD display, making it easier to show calculated results, sensor readings, or numerical information in embedded system applications.

Code 4.4 Formatted Data Display

```
/*
  This program demonstrates how to display formatted data on an I2C LCD.

  - Shows similarity between lcd.print() and Serial.print() functions.
  - Prints a floating-point number with specified decimal precision.
  - Prints an integer value in hexadecimal format using format specifiers.

  The example calculates the sum of two floating-point numbers and
  displays it with 3 decimal places. It also shows how an integer
  can be displayed in different number systems (HEX in this case).
*/


// Include Library Files
#include<LiquidCrystal_I2C.h>


// Declaring LCD_I2C based Object
LiquidCrystal_I2C lcd(0x3F, 16, 2);


void setup()
{
  lcd.init(); //initializes lcd
  lcd.backlight(); //this function brings light at the back of LCD

  lcd.setCursor(0, 0); // Top Line
  float num1 = 10.123;
  float num2 =  5.222;
  float sum = num1 + num2; // 15.345
  lcd.print(sum, 3); // '3' digits after decimal

  lcd.setCursor(0, 1); // Bottom Line
  lcd.print(15, HEX); // 'F' (Showing number in HEX format)
}

void loop() {

}
```

User Input from Serial Monitor to LCD Display

User interaction with an Arduino can be achieved by taking input from the Serial Monitor and displaying it on an LCD. The Serial Monitor acts as a simple interface where the user can type text or numbers, which the Arduino reads using serial communication functions like `Serial.parseInt()` or `Serial.readString()`. Once the data is received, it can be sent to an LCD using `lcd.print()` to visually display the information. This method provides a simple way to combine user input and real-time visual feedback, making it useful for interactive projects, menus, messages, or data display in embedded systems.

Code 4.5 Serial Display

```
/*
  This sketch shows how to take input (String & Numbers) from serial
  monitor and display it in the LCD.

  In this code wee would ask from user to input their Name (String) &
  Weight (Number) and show the corresponding message in LCD.
*/

#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x3F, 16, 2); //Slave Address, Column, Row

void setup() {

  // Serial Display
  Serial.begin(9600);

  lcd.init();
  lcd.backlight();

  // String (Name) Input

  lcd.setCursor(0, 0);
  lcd.print("Name: ");

  while (Serial.available() == 0) {}
  lcd.print(Serial.readString());


  // Number (Weight) Input

  lcd.setCursor(0, 1);
  lcd.print("Weight: ");

  while (Serial.available() == 0) {}
  lcd.print(Serial.parseInt());
  lcd.print(" kg");
}

void loop() {

}
```

Scrolling Display

The scrolling display feature on an I2C LCD allows messages to move smoothly across the display, creating a dynamic and visually appealing effect. I2C LCD library provides built-in functions like `lcd.scrollDisplayLeft()` and `lcd.scrollDisplayRight()` to shift the entire content of the screen one column at a time in either direction. This is especially useful when the text is longer than the available display width, as it enables all information to be seen without truncation. By controlling the scroll speed with delays, users can create readable animations or marquee effects, making the LCD output more interactive and engaging. Scrolling text is commonly used for notifications, greetings, sensor readings, or other messages that need to be continuously updated or emphasized on embedded systems. Additionally, combining scrolling with cursor positioning and dynamic content allows for more creative displays, such as running status messages or real-time updates, enhancing both functionality and user experience.

Code 4.6 Rotating Text

```
/*
  This sketch prints a user input text to the LCD and uses the
  string length to determine last position to scroll the text
  so that it is visible at the end of scrolling. After scrolling
  from left to right it moves into the next line (Right Justified).
  Then it scrolls from Right to Left position so that ultimately
  we would get the rotating text effect using scroll() method.
*/

// include the library code:
#include <LiquidCrystal_I2C.h>

LiquidCrystal_I2C lcd(0x3F, 16, 2); //Slave Address, Column, Row

String msg; // String Input Message
int len; // Length of the String

void setup() {

  // Initializing Serial Monitor
  Serial.begin(9600);

  // Initializing the LCD display
  lcd.init();
  lcd.backlight();

  Serial.print("Input Message: ");

  // Taking Input Message from User
  while (Serial.available() == 0) {}
  msg = Serial.readString();// Input String
  len = msg.length(); // Finding the length of String
  Serial.println(msg); // Printing the Message in Serial Monitor

}
```

```
void loop() {

  // -------  Left to Right Scroll -------

  // Cursor at Top Left Position
  lcd. setCursor(0, 0);
  lcd.print(msg); // Printing the message in LCD Display
  delay(500);

  // Moving Left to Right in Top Row
  for (int i = 0; i < 16 - len; i++)
  {
    lcd.scrollDisplayRight();
    delay(500); // Adding Delay to see the message clearly
  }



  // Clearing the Message
  lcd.clear();
  delay(500);


  // -------  Right to Left Scroll -------

  // Cursor at Bottom Right Position
  lcd. setCursor(16 - len, 1);
  lcd.print(msg); // Printing the message in LCD Display
  delay(500);

  // Moving Right to Left in Bottom Row
  for (int i = 16 - len; i > 0; i--)
  {
    lcd.scrollDisplayLeft();
    delay(500); // Adding Delay to see the message clearly
  }

  // Clearing the Message
  lcd.clear();
  delay(500);
}
```

Chapter

5 Seven Segment Display

A Single Display

A 7-segment display is a simple and widely used electronic display device for representing numerical digits and a limited set of characters. The display is made up of seven individual light-emitting segments arranged in the shape of the digit "8". When a segment is forward biased through an appropriate current-limiting resistor, it emits light. The combination of illuminated segments determines the digit shown on the display. By selectively turning these segments ON or OFF, different numeric digits are displayed. It is commonly used in digital systems as they are simple and inexpensive, easy to interface with digital circuits, and provide high brightness and good readability. They are reliable and suitable for displaying numeric information in many electronic applications. These type of displays are widely used in digital clocks, calculators, counters, timers, voltmeters, and other devices where numerical information needs to be displayed clearly and efficiently.

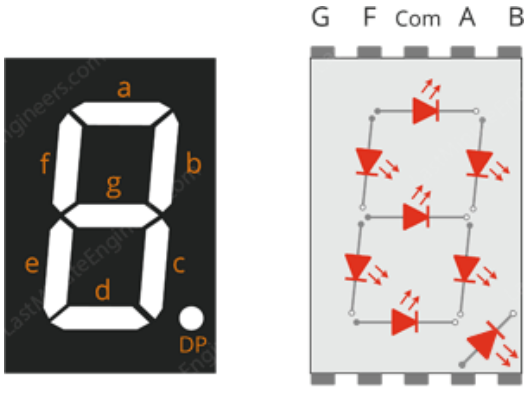


Figure 5.1: Seven Segment Display

Based on internal wiring, 7-segment displays are classified into two main types:

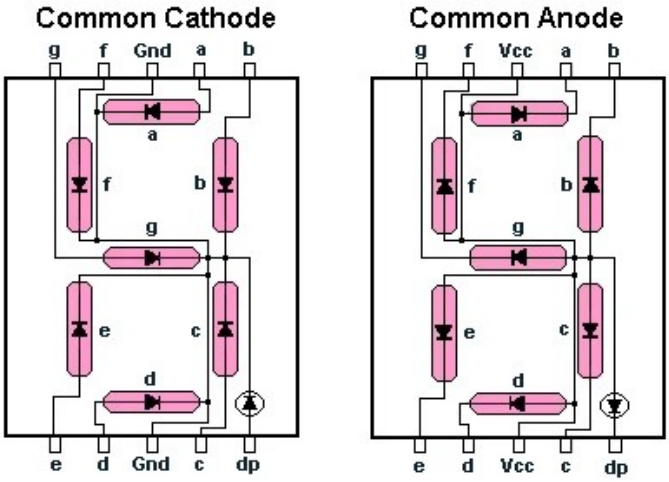


Figure 5.2: Internal Connection Diagram

- **Common Cathode (CC):** All cathodes of the LEDs are connected together and usually tied to ground. A segment turns ON when a logic HIGH voltage is applied to its anode.
- **Common Anode (CA):** All anodes of the LEDs are connected together and usually tied to the supply voltage. A segment turns ON when a logic LOW voltage is applied to its cathode.

Table 5.1: Comparison Table

Feature	**Common Cathode (CC)**	**Common Anode (CA)**
Common pin connection	GND	VCC
Segment ON condition	Logic HIGH (1)	Logic LOW (0)
Logic nature	Non-inverted	Inverted

Due to their simple working principle and reliability, 7-segment displays are extensively used in digital clocks, calculators, counters, timers, measuring instruments, and other electronic devices where numeric information needs to be displayed in a clear and straightforward manner. However, they cannot display full text or complex characters and are limited to numeric and very simple symbols. They consume more power than LCD and become bulky when used for displaying large amounts of data.

A standard 7-segment display consists of multiple LEDs arranged in a specific pattern and interconnected internally. Its main structural components are:

- Seven LED segments labeled as **a, b, c, d, e, f, g**
- One additional LED segments labeled as **decimal point (DP)** or simply **p**
- Two **common terminals** (common anode or common cathode) shared by all segments

Each segment is an individual LED, while the common terminal provides a single connection point for either all anodes or all cathodes of the LEDs.

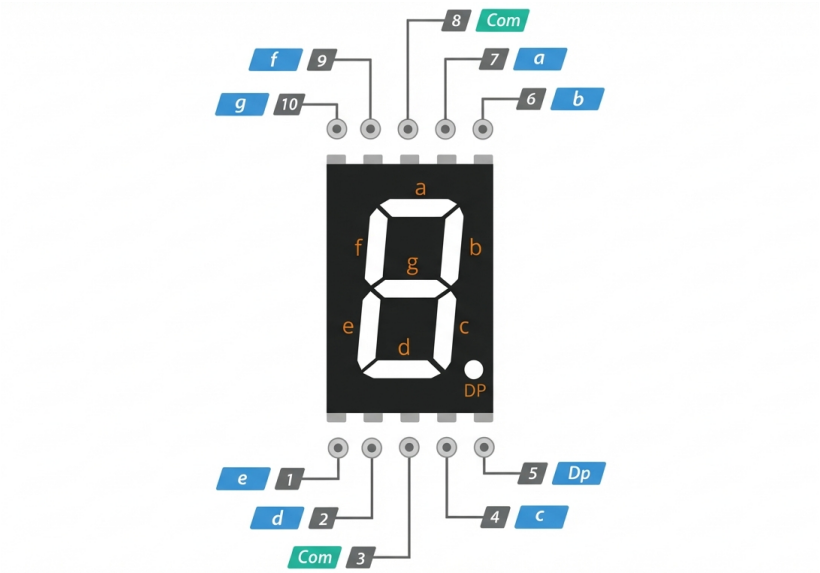


Figure 5.3: Pinouts

Now, we need to interface the 7-segment display with the Arduino Uno. The pin connections are listed in the following table, and the wiring diagram is shown in the following figures.

Table 5.2: Connection between CC 7SD & Arduino

7SD	Arduino	7SD	Arduino
a	2	b	3
c	4	d	5
e	6	f	7
g	8	p	9
Common	GND via Resistor (220 Ω)		

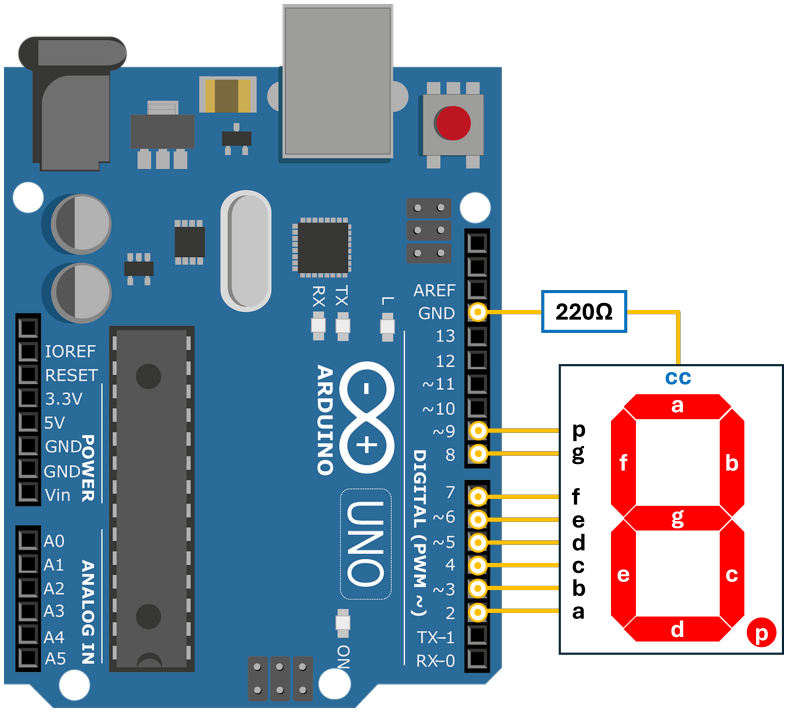


Figure 5.4: Arduino Connection with 7SD

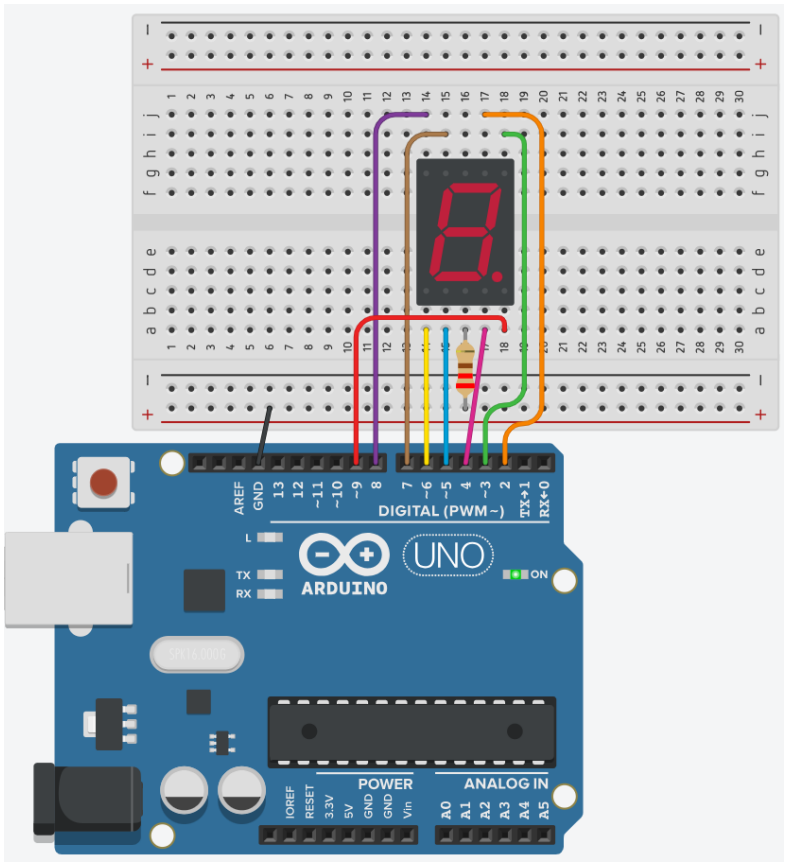
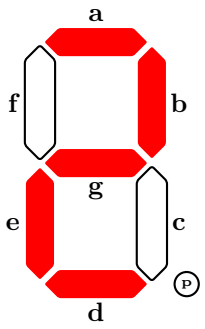


Figure 5.5: Breadboard connection of Figure 5.4

Basic Implementation

A seven-segment display can be viewed as a set of eight LEDs connected together in a single unit. In a common cathode configuration, the shared terminal is connected to ground to allow the segments to turn ON when a HIGH signal is applied. Now, each segment can be controlled like a normal LED using digital output pins. Inside the `setup()`, pins 2 to 9 are configured as output pins using `pinMode()` command, each representing one segment (a–p). Inside the `loop()`, the `digitalWrite()` command control the ON/OFF state of each segment by setting them HIGH (ON) or LOW (OFF). The selected combination of HIGH signals for segments `a`, `b`, `d`, `e`, and `g`, while keeping the remaining segments OFF, produces the digit **2** on the display.



Code 5.1 Basic

```
/*
   In this code, we will learn how to show some value
   in a Common Cathode (CC) 7-Segment Display.

   This is a very basic level code where we will
   show "2" in 7SD (Here 'a' -> 2 ... 'p' -> 9)
*/

void setup() {
  pinMode(2, OUTPUT); // 'a'
  pinMode(3, OUTPUT); // 'b'
  pinMode(4, OUTPUT); // 'c'
  pinMode(5, OUTPUT); // 'd'
  pinMode(6, OUTPUT); // 'e'
  pinMode(7, OUTPUT); // 'f'
  pinMode(8, OUTPUT); // 'g'
  pinMode(9, OUTPUT); // 'p'
}

void loop() {
  digitalWrite(2, HIGH);  // 'a' = 1
  digitalWrite(3, HIGH);  // 'b' = 1
  digitalWrite(4, LOW);   // 'c' = 0
  digitalWrite(5, HIGH);  // 'd' = 1
  digitalWrite(6, HIGH);  // 'e' = 1
  digitalWrite(7, LOW);   // 'f' = 0
  digitalWrite(8, HIGH);  // 'g' = 1
  digitalWrite(9, LOW);   // 'p' = 0
}
```

Hexadecimal Digits

A 7-segment Display can represent all hexadecimal digits (0 - F) by lighting appropriate combinations of its seven segments. The decimal digits 0 - 9 are displayed normally, while the letters A, b, C, d, E, and F are commonly used to represent hexadecimal values 10 - 15 because their shapes are clearly recognizable on a 7-segment Display. This allows a 7-segment display to show all 16 hexadecimal symbols in a simple and compact numeric interface.

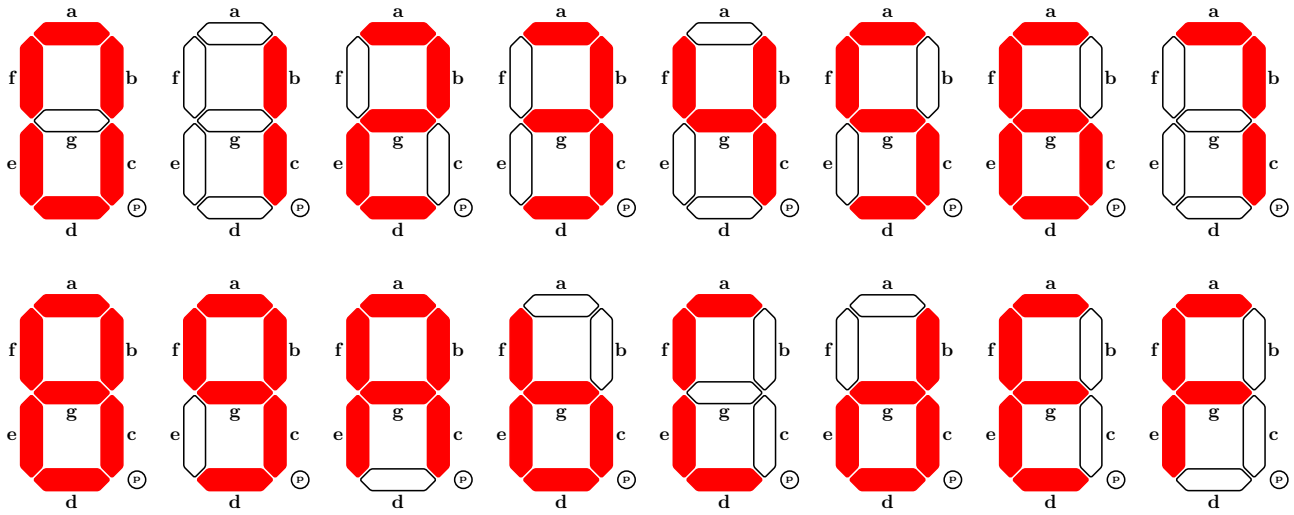
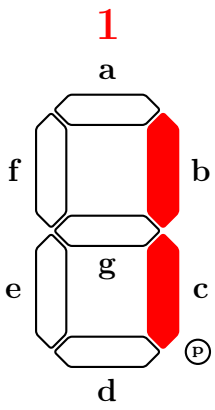
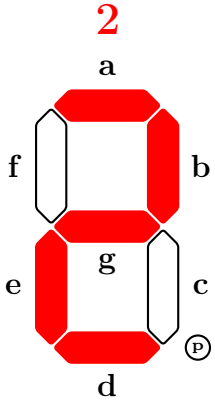


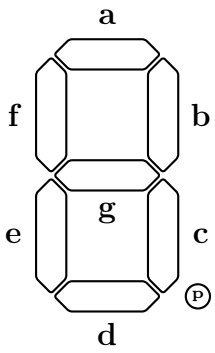
Figure 5.6: HEX Digits



p	g	f	e	d	c	b	a	Base
0	0	0	0	0	1	1	0	BIN
0				6				HEX

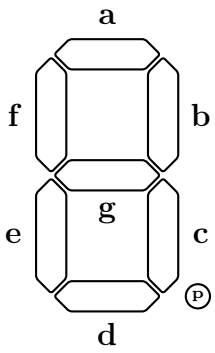


p	g	f	e	d	c	b	a	Base
0	1	0	1	1	0	1	1	BIN
5				B				HEX



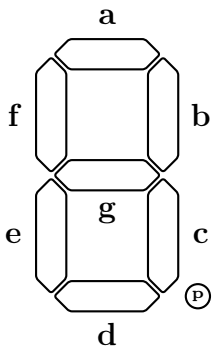
p	g	f	e	d	c	b	a

. . . . . .



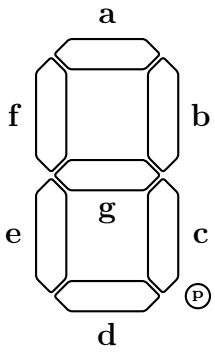
p	g	f	e	d	c	b	a

. . . . . .



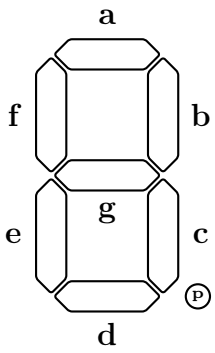
p	g	f	e	d	c	b	a

. . . . . .



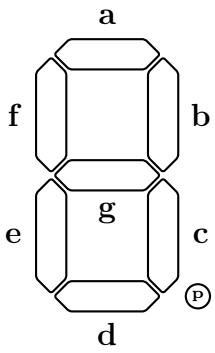
p	g	f	e	d	c	b	a

. . . . . .



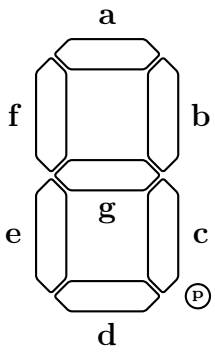
p	g	f	e	d	c	b	a

. . . . . .



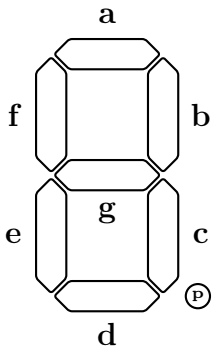
p	g	f	e	d	c	b	a

. . . . . .



p	g	f	e	d	c	b	a

. . . . . .



p	g	f	e	d	c	b	a

. . . . . .

Common Cathode Hex Code Table

A hexadecimal code table for a 7-segment display provides a quick and systematic way to determine the segment activation pattern for different characters or numbers. Instead of manually checking which segments should be ON or OFF each time, this table simplifies the process by directly giving the corresponding hex values. It reduces design time, minimizes errors in coding, and helps ensure consistent display control in digital and microcontroller-based applications.

Table 5.3: Common Cathode (CC) Code

Value	p	g	f	e	d	c	b	a	**Hex**
0	0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	0	1	1	0	0x06
2	0	1	0	1	1	0	1	1	0x5B
3	0	1	0	0	1	1	1	1	0x4F
4	0	1	1	0	0	1	1	0	0x66
5	0	1	1	0	1	1	0	1	0x6D
6	0	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	0	1	1	1	0x07
8	0	1	1	1	1	1	1	1	0x7F
9	0	1	1	0	1	1	1	1	0x6F
A	0	1	1	1	0	1	1	1	0x77
b	0	1	1	1	1	1	0	0	0x7C
C	0	0	1	1	1	0	0	1	0x39
d	0	1	0	1	1	1	1	0	0x5E
E	0	1	1	1	1	0	0	1	0x79
F	0	1	1	1	0	0	0	1	0x71

Implementation using HEX Code & Bit Extraction

In this program, the digit is displayed using a hexadecimal (HEX) code instead of manually setting each segment. For common cathode configuration, a predefined byte value (`0x5B`) representing the digit **2** is stored, where each bit corresponds to a specific segment in the order (p g f e d c b a). This value is taken from the Common Cathode Hex Code Table for the 7-segment display shown in Table 5.3. Then the `bitRead()` function extracts individual bits from the stored code, and each extracted bit is sent to the corresponding segment through `digitalWrite()`. A bit value of 1 (HIGH) turns the segment ON, while 0 (LOW) turns it OFF. This method simplifies the control process by representing the entire segment pattern with a single hexadecimal value and automatically distributing the required signals to produce the digit **2** on the display.

Code 5.2 Bit Read

```
/*
   In this we will learn how to show some value
   in a Common Cathod (CC) 7-Segment Display.

   Here, we will use "bitRead()" function to
   show "2" in 7SD (Here 'a' -> 2 ... 'p' -> 9)
*/


// Bit order: (p g f e d c b a)
byte Code = 0x5B; // HEX Code of "2" (Binary: 0101_1011)


void setup() {
  pinMode(2, OUTPUT); // 'a'
  pinMode(3, OUTPUT); // 'b'
  pinMode(4, OUTPUT); // 'c'
  pinMode(5, OUTPUT); // 'd'
  pinMode(6, OUTPUT); // 'e'
  pinMode(7, OUTPUT); // 'f'
  pinMode(8, OUTPUT); // 'g'
  pinMode(9, OUTPUT); // 'p'
}


void loop() {
  digitalWrite(2, bitRead(Code, 0)); // 'a' = 1
  digitalWrite(3, bitRead(Code, 1)); // 'b' = 1
  digitalWrite(4, bitRead(Code, 2)); // 'c' = 0
  digitalWrite(5, bitRead(Code, 3)); // 'd' = 1
  digitalWrite(6, bitRead(Code, 4)); // 'e' = 1
  digitalWrite(7, bitRead(Code, 5)); // 'f' = 0
  digitalWrite(8, bitRead(Code, 6)); // 'g' = 1
  digitalWrite(9, bitRead(Code, 7)); // 'p' = 0
}
```

Now, to display other hexadecimal digits on the 7-segment display, simply refer to the Common Cathode Hex Code Table and use the corresponding value. The selected code will automatically generate the required segment pattern to display the desired digit.

Implementation using Array

In this implementation, an array is used to store the pin numbers connected to each segment of the seven-segment display, which improves code readability and simplifies pin management. Instead of repeatedly writing individual pin numbers inside both `setup()` and `loop()`, all segment pins are grouped into a single array, so any change in hardware connection requires modification at only one place in the program. Now, each segment can be configured and controlled through indexed array elements, making the code more organized and easier to maintain. An important point to note is that the pin configuration and hexadecimal code follow opposite directions, but this is done intentionally for proper indexing. The pin array is arranged in the order `(a b c d e f g p)`, where `Pin[0]` represents the leftmost element corresponding to segment `a`. In contrast, the hexadecimal code follows the bit order `(p g f e d c b a)`, where `bitRead(Code, 0)` extracts the rightmost bit corresponding to segment `a`. This reverse arrangement ensures that the same index value refers to both the segment pin and its corresponding bit value, allowing direct matching between the array index and segment control. This approach simplifies the implementation in Arduino programming by making the code more consistent and easier to manage. Finally, each extracted bit is then applied to the respective segment through `digitalWrite()`, allowing efficient and structured control of the display while reducing redundancy in the program.

Code 5.3 Array

```
/*
   Here, we will use Array for Segment Pins to show "2"
*/

// Pin order: (a b c d e f g p)
byte Pin[] = {2, 3, 4, 5, 6, 7, 8, 9};

// Bit order: (p g f e d c b a)
byte Code = 0x5B; // HEX Code of "2" (Binary: 0101_1011)

void setup() {
  pinMode(Pin[0], OUTPUT); // 'a'
  pinMode(Pin[1], OUTPUT); // 'b'
  pinMode(Pin[2], OUTPUT); // 'c'
  pinMode(Pin[3], OUTPUT); // 'd'
  pinMode(Pin[4], OUTPUT); // 'e'
  pinMode(Pin[5], OUTPUT); // 'f'
  pinMode(Pin[6], OUTPUT); // 'g'
  pinMode(Pin[7], OUTPUT); // 'p'
}

void loop() {
  digitalWrite(Pin[0], bitRead(Code, 0)); // 'a' = 1
  digitalWrite(Pin[1], bitRead(Code, 1)); // 'b' = 1
  digitalWrite(Pin[2], bitRead(Code, 2)); // 'c' = 0
  digitalWrite(Pin[3], bitRead(Code, 3)); // 'd' = 1
  digitalWrite(Pin[4], bitRead(Code, 4)); // 'e' = 1
  digitalWrite(Pin[5], bitRead(Code, 5)); // 'f' = 0
  digitalWrite(Pin[6], bitRead(Code, 6)); // 'g' = 1
  digitalWrite(Pin[7], bitRead(Code, 7)); // 'p' = 0
}
```

Simplified Implementation using Loop

In this program, the control of the seven-segment display is further simplified by using a loop structure to handle both pin initialization and segment control. Instead of writing multiple repetitive statements for each segment, a `for` loop is used in the `setup()` to configure all segment pins stored in the array as output pins, and another `for` loop in the `loop()` is used to send the corresponding bit values to each segment. The `bitRead()` function extracts each bit from the HEX code and directly applies it to the matching pin through indexed array access. This approach makes the program shorter, more structured, and easier to modify, as any change in the number of segments or pin configuration can be handled efficiently by adjusting the loop or array rather than rewriting multiple statements. It also improves code readability by clearly showing the sequential relationship between segment pins and their corresponding bit values.

Code 5.4 Loop

```
/*
   In this we will learn how to show some value
   in a Common Cathode (CC) 7-Segment Display.

   Here, we will use a "for" loop to show "2" in 7SD.
*/


// Pin order: (a b c d e f g p)
byte Pin[] = {2, 3, 4, 5, 6, 7, 8, 9};


// Bit order: (p g f e d c b a)
byte Code = 0x5B; // HEX Code of "2" (Binary: 0101_1011)


void setup() {

  // Setting Pin Modes
  for (int i = 0; i < 8; i++)
  {
    pinMode(Pin[i], OUTPUT);
  }

}

void loop() {

  // Turning segments ON/OFF
  for (int i = 0; i < 8; i++)
  {
    digitalWrite(Pin[i], bitRead(Code, i));
  }

}
```

As shown in this code, it can be easily understood that the same task can be performed with only a few lines by using loop statements, which also makes the program easier to maintain.

Modular Implementation using Custom Function

This program demonstrates a more streamlined and modular approach to controlling a seven-segment display by defining a custom function to handle the segment driving process. Instead of placing the segment control logic directly inside the `loop()` function, the operations are encapsulated within a user-defined function `displayDigit()`, which takes a byte pattern as input and updates all segments accordingly. In the `setup()` function, a `for` loop configures all segment pins stored in the array as output pins, while the `loop()` function simply calls the custom function with the required hexadecimal code. Inside the function, another loop extracts each bit from the given pattern using `bitRead()` and sends the corresponding values to the segment pins. This modular design improves code organization, reusability, and readability, allowing different digits or patterns to be displayed easily by passing different values to the function without modifying the main program logic.

Code 5.5 Function

```
/*
   In this we will learn how to show some value
   in a Common Cathode (CC) 7-Segment Display.

   Here, we will use a Custom Function to show "2" in 7SD.
*/

// Pin order: (a b c d e f g p)
byte Pin[] = {2, 3, 4, 5, 6, 7, 8, 9};

// Bit order: (p g f e d c b a)
byte Code = 0x5B; // HEX Code of "2" (Binary: 0101_1011)

void setup() {

  for (int i = 0; i < 8; i++)
  {
    pinMode(Pin[i], OUTPUT);
  }

}

void loop() {

  displayDigit(Code);

}

// Custom Function for 7SD
void displayDigit(byte pattern) {

  for (int i = 0; i < 8; i++)
  {
    digitalWrite(Pin[i], bitRead(pattern, i));
  }

}
```

The structured and reusable approach demonstrated in the previous program provides a foundation for more advanced applications, enabling different display sequences or custom characters to be implemented easily by modifying or expanding the stored segment patterns. For example, the following program extends the modular seven-segment display control method by using it as a base framework to display all HEX digits (0–F) sequentially. An array is used to store the segment patterns for all 16 hexadecimal values. In the `loop()` function a loop iterates through the array of HEX codes and sends each pattern to the display one by one with a fixed delay.

Code 5.6 Show All

```
/* Here, we will show 0 to F one by one in the 7SD. */

// Pin order: (a b c d e f g p)
int Pin[] = { 2, 3, 4, 5, 6, 7, 8, 9 };

// Bit order: (p g f e d c b a)
byte Code[16] = {
  0x3F,  // 0
  0x06,  // 1
  0x5B,  // 2
  0x4F,  // 3
  0x66,  // 4
  0x6D,  // 5
  0x7D,  // 6
  0x07,  // 7
  0x7F,  // 8
  0x6F,  // 9
  0x77,  // A
  0x7C,  // b
  0x39,  // C
  0x5E,  // d
  0x79,  // E
  0x71   // F
};

void setup() {
  for (int i = 0; i < 8; i++) {
    pinMode(Pin[i], OUTPUT);
  }
}

void loop() {
  // Showing ALL 16 HEX digits one by one
  for (int i = 0; i <= 15; i++) {
    displayDigit(Code[i]);
    delay(1000);
  }
}

void displayDigit(byte pattern) {
  for (int i = 0; i < 8; i++) {
    digitalWrite(Pin[i], bitRead(pattern, i));
  }
}
```

In this program, the modular seven-segment display framework developed earlier is further extended to allow dynamic control through user input from the Serial Monitor. The program waits for the user input and then applies the `constrain()` function to ensure that the value remains within the valid hexadecimal range of 0 to 15. This validated value is used as an index to access the corresponding segment pattern from the stored code array, which is then passed to the display function to show the appropriate digit on the seven-segment display. This implementation demonstrates how real-time user interaction can be integrated into the display system, enabling flexible and dynamically controlled digit selection in Arduino-based applications.

Code 5.7 User Input

```
/*
   Here, we will take user input through Serial Monitor
   and show the corresponding digit in the 7SD (0 - 15).
*/

// Pin order: (a b c d e f g p)
int Pin[] = {2, 3, 4, 5, 6, 7, 8, 9};

// Bit order: (p g f e d c b a)
byte Code[16] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07,
                 0x7F, 0x6F, 0x77, 0x7C, 0x39, 0x5E, 0x79, 0x71};

void setup() {

  Serial.begin(9600); // Initializing Serial Monitor

  // Setting Pin Modes
  for (int i = 0; i < 8; i++){
    pinMode(Pin[i], OUTPUT);
  }

}

void loop() {

  // Taking Integer Input
  Serial.print("n = ");
  while (Serial.available() == 0) {}
  int n = Serial.parseInt();
  Serial.println(n);
  n = constrain(n, 0, 15); // limiting value in [0,15]

  displayDigit(Code[n]); // Calling the custom function
}

void displayDigit(byte pattern) {

  for (int i = 0; i < 8; i++){
    digitalWrite(Pin[i], bitRead(pattern, i));
  }

}
```

B 4×7 Segment Display

A 4×7 segment display consists of four individual seven-segment digits arranged side by side to display multi-digit numerical values. Each digit is made up of seven LED segments (labeled a – g) and an optional decimal point. In Arduino or microcontroller projects, the display is commonly driven using multiplexing to reduce the number of I/O pins. Depending on the wiring, the display can be either common anode (CA) or common cathode (CC). This type of display is widely used in digital clocks, counters, timers, and measurement instruments.

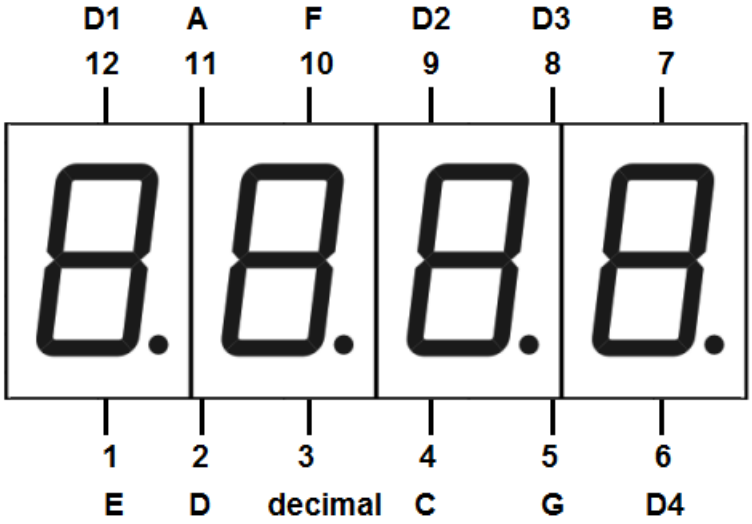


Figure 5.7: 4×7SD Pinouts

Connection with Arduino

A 4-digit 7-segment display has 12 pins: 8 segment pins (a–g and p) and 4 digit-select pins (common cathodes for each digit). The detailed pin connections are listed in the following table.

Table 5.4: Connection between 4×7SD & Arduino

4×7SD	Arduino	4×7SD	Arduino	4×7SD	Arduino
a	2	b	3	D1	A0 via 220Ω
c	4	d	5	D2	A1 via 220Ω
e	6	f	7	D3	A2 via 220Ω
g	8	p	9	D4	A3 via 220Ω

In this setup, the segment pins a–p are connected to Arduino UNO digital pins 2–9, respectively. The four digit-select pins, which control which digit is active, are connected to analog pins A0–A3. Each segment digit-select is connected through a current-limiting resistor to protect the LEDs inside the display from excessive current.

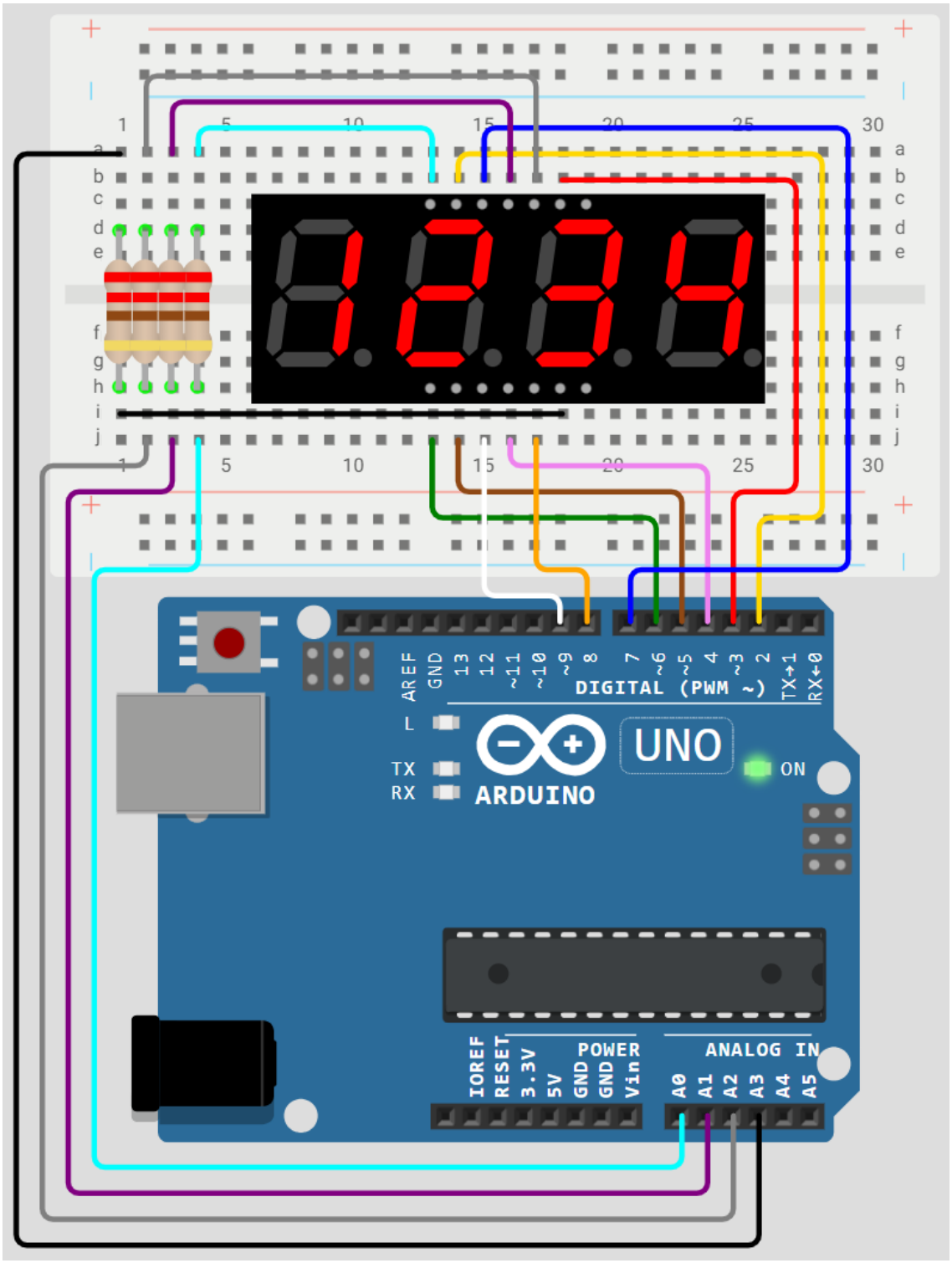


Figure 5.8: 4×7SD connection with Arduino

Internal Connection

All corresponding segment pins (a–p) of the four digits are internally connected together to form shared segment lines. Each digit, however, has its own separate common terminal (either common anode or common cathode), which is used to select that digit. Because of this internal wiring, the display is driven using multiplexing: the controller applies the required segment pattern on the shared segment lines and then enables one digit's common pin at a time, very quickly switching between digits so that all appear continuously ON to the human eye. This reduces the number of I/O pins needed and allows independent numbers to be shown on each digit, even though the segment lines are common.

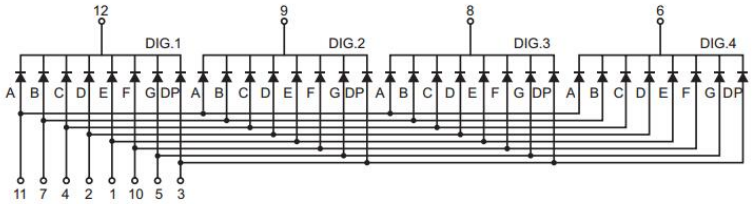


Figure 5.9: 4×7SD Internal Connection

All Possible States

Seven-segment display consists of seven LEDs if we ignore the decimal point LED. By turning different combinations of these segments ON or OFF, all numeric digits and several alphabetic characters can be formed. Any other combination of the seven segments represents additional symbols or undefined patterns, giving a total of $2^7 = 128$ possible states, although only a limited subset is practically used for readable characters.

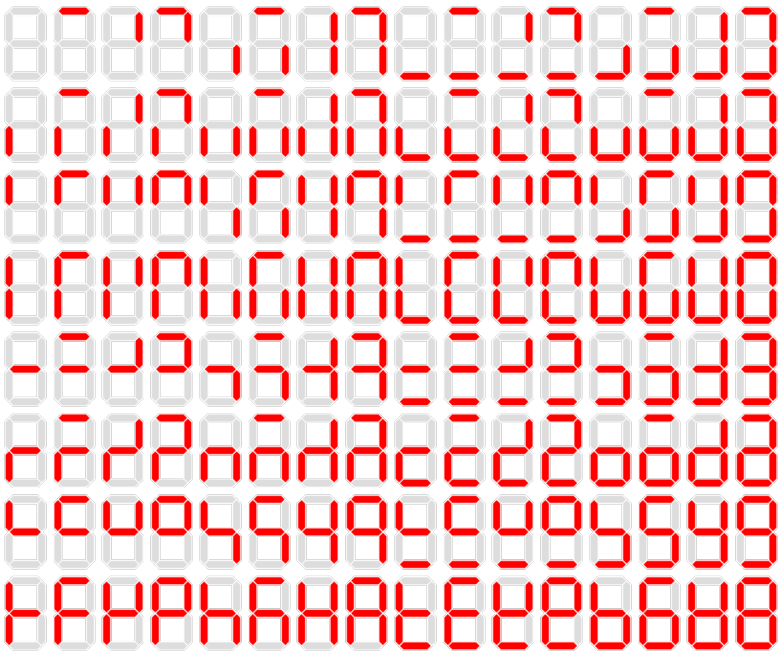
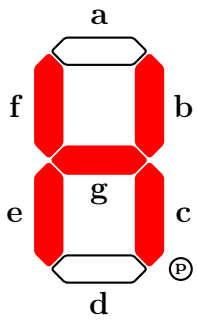


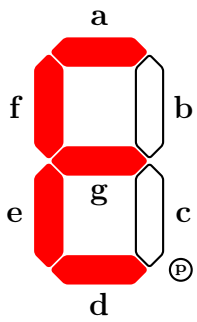
Figure 5.10: All Possible States

H



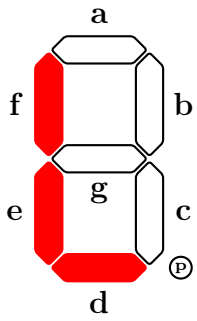
p	g	f	e	d	c	b	a

E



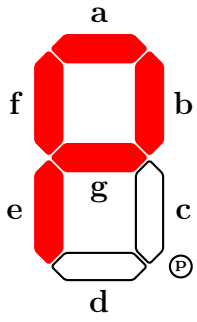
p	g	f	e	d	c	b	a

L



p	g	f	e	d	c	b	a

P



p	g	f	e	d	c	b	a

Code 5.8 4-Character Word Display

```
/*
   4x7 Segment Display (Common Cathode) shows "HELP" using multiplexing
*/

// Segment pins: (a b c d e f g p)
int segPin[] = {2, 3, 4, 5, 6, 7, 8, 9};

// Digit select pins (D1 D2 D3 D4)
int digPin[] = {A0, A1, A2, A3};

// Display buffer (Digit1, Digit2, Digit3, Digit4)
byte displayBuf[4] = {0x76, 0x79, 0x38, 0x73}; // "H", "E", "L", "P"

void setup() {

  // Segment pins
  for (int i = 0; i < 8; i++) {
    pinMode(segPin[i], OUTPUT);
  }

  // Digit pins
  for (int d = 0; d < 4; d++) {
    pinMode(digPin[d], OUTPUT);
  }
}

void loop() {

  // Refresh display continuously
  for (int d = 0; d < 4; d++) {
    showDigit(d, displayBuf[d]);
  }
}

// Show digit one by one
void showDigit(int digit, byte pattern) {

  // Turn all digits OFF
  for (int d = 0; d < 4; d++) {
    digitalWrite(digPin[d], HIGH);
  }

  // Apply segment pattern
  for (int i = 0; i < 8; i++) {
    digitalWrite(segPin[i], bitRead(pattern, i));
  }

  // Enable selected digit
  digitalWrite(digPin[digit], LOW);

  delay(3);   // multiplex delay
}
```

Multi-Digit Number Display

To display a 4-digit integer or floating-point number on a 4-digit 7-segment display, the program first stores the hexadecimal segment codes corresponding to all decimal digits. For integer values, the number is decomposed into individual digits so that each digit can be shown on a separate display position. This is done using division and modulus operations. For example, in the number 1234, the thousands, hundreds, tens, and units digits are extracted sequentially and stored in a buffer array, where each element represents the digit to be displayed on a specific 7-segment position.

For floating-point numbers, an additional step is required to determine the correct placement of the decimal point. The program checks the magnitude of the input value and multiplies it by an appropriate power of 10 so that it can be converted into an integer representation suitable for digit extraction. At the same time, a variable is used to record the position where the decimal point should appear on the display. For example, if the value is less than 10, it is multiplied by 1000 so that three decimal places can be shown, whereas larger values require smaller scaling factors. This approach allows both integer and floating-point numbers to be displayed correctly on the 4-digit 7-segment display while maintaining proper decimal point positioning.

Code 5.9 4-Digit Decimal Number Display

```
/*
   4x7 Segment Display (Common Cathode)
   Displays 4 decimal digits using multiplexing
*/


// Segment pins: (a b c d e f g p)
int segPin[] = {2, 3, 4, 5, 6, 7, 8, 9};


// Digit select pins (D1 D2 D3 D4)
int digPin[] = {A0, A1, A2, A3};


// Hex codes (p g f e d c b a)
byte Code[16] = {
  0x3F, 0x06, 0x5B, 0x4F,
  0x66, 0x6D, 0x7D, 0x07,
  0x7F, 0x6F, 0x77, 0x7C,
  0x39, 0x5E, 0x79, 0x71
};


// Display buffer
byte displayBuf[4];


// Number to display
unsigned int number = 1234;
```

```
void setup() {

  // Segment pins
  for (int i = 0; i < 8; i++)
  {
    pinMode(segPin[i], OUTPUT);
  }

  // Digit pins
  for (int d = 0; d < 4; d++)
  {
    pinMode(digPin[d], OUTPUT);
  }

  // Convert number → digit array
  displayBuf[0] = number / 1000;        // Digit 1 (MSB)
  displayBuf[1] = (number / 100)  % 10; // Digit 2
  displayBuf[2] = (number / 10)   % 10; // Digit 3
  displayBuf[3] = number % 10;          // Digit 4 (LSB)

}


void loop() {

  // Refresh display continuously
  for (int d = 0; d < 4; d++)
  {
    showDigit(d, Code[displayBuf[d]]);
  }

}


// Show digit one by one
void showDigit(int digit, byte pattern) {

  // Turn all digits OFF
  for (int d = 0; d < 4; d++)
  {
    digitalWrite(digPin[d], HIGH);
  }

  // Apply segment pattern
  for (int i = 0; i < 8; i++)
  {
    digitalWrite(segPin[i], bitRead(pattern, i));
  }

  // Enable selected digit
  digitalWrite(digPin[digit], LOW);

  delay(3);   // multiplex delay

}
```

Code 5.10 4-Digit Float Number Display

```
/*
   4x7 Segment Display (Common Cathode)
   Displays 4 digits Float Number using multiplexing

   First we will convert the float into integer and
   in that process would determine the dot point position
   and we would use the position to set the "p" segment value.
*/


// Segment pins: (a b c d e f g p)
int segPin[] = {2, 3, 4, 5, 6, 7, 8, 9};


// Digit select pins (D1 D2 D3 D4)
int digPin[] = {A0, A1, A2, A3};


// Hex codes (p g f e d c b a)
byte Code[16] = {
  0x3F, 0x06, 0x5B, 0x4F,
  0x66, 0x6D, 0x7D, 0x07,
  0x7F, 0x6F, 0x77, 0x7C,
  0x39, 0x5E, 0x79, 0x71
};


// Display buffer
byte displayBuf[4];


// Float Number to display
float val = 23.75;
unsigned int number; // Integer Number
unsigned int n; // Dot Position


void setup() {

  // Segment pins
  for (int i = 0; i < 8; i++)
  {
    pinMode(segPin[i], OUTPUT);
  }


  // Digit pins
  for (int d = 0; d < 4; d++)
  {
    pinMode(digPin[d], OUTPUT);
  }
```

```
  // Converting Float Number into Integer
  // and Finding Decimal Point Position

  if (val < 10) {
    number = val * 1000;
    n = 0;
  }
  else if (val < 100) {
    number = val * 100;
    n = 1;
  }
  else if (val < 1000) {
    number = val * 10;
    n = 2;
  }
  else{
    number = val;
    n = 3;
  }

  // Convert number → digit array (inside setup)
  displayBuf[0] = number / 1000;        // Digit 1 (MSB)
  displayBuf[1] = (number / 100)  % 10; // Digit 2
  displayBuf[2] = (number / 10)   % 10; // Digit 3
  displayBuf[3] = number % 10;          // Digit 4 (LSB)

}


void loop() {
  for (int d = 0; d < 4; d++) {
    showDigit(d, (Code[displayBuf[d]] | (d == n) * 0x80));
  }
}


// Show digit one by one
void showDigit(int digit, byte pattern) {

  // Turn all digits OFF
  for (int d = 0; d < 4; d++) {
    digitalWrite(digPin[d], HIGH);
  }

  // Apply segment pattern
  for (int i = 0; i < 8; i++) {
    digitalWrite(segPin[i], bitRead(pattern, i));
  }

  // Enable selected digit
  digitalWrite(digPin[digit], LOW);

  delay(3);   // multiplex delay
}
```

Chapter

6 Analog Operation

A Analog Read

In Arduino Uno it is possible to measure a continuous (analog) voltage present on an analog input pin by converting it into a corresponding digital value using the built-in Analog-to-Digital Converter (ADC) of the ATmega328P microcontroller. Since microcontrollers operate digitally, they cannot directly process analog voltages; therefore, the ADC acts as an interface between the analog world and the digital system. The Arduino Uno employs a 10-bit successive approximation ADC, which samples the input voltage, compares it against a reference voltage, and produces a numerical value proportional to the input level. The returned digital value represents the ratio between the input voltage and the selected reference voltage, enabling the microcontroller to interpret sensor outputs. The internal operation of the ADC process is described below and the important parameters related to the ADC are summarized in the following table.

Internal Operation

① The selected analog pin is connected to the ADC through an internal multiplexer.

② The chosen reference voltage is applied to the ADC.

③ The sample-and-hold capacitor captures the input voltage.

④ The ADC conversion process is initiated.

⑤ The successive approximation logic compares input voltage with internal reference levels.

⑥ A binary search determines the closest 10-bit digital equivalent.

⑦ The conversion completes after 13 ADC clock cycles.

⑧ The digital result is stored in the ADC data registers.

⑨ The final value (0 – 1023) is returned to the program.

Table 6.1: ADC Parameters

Parameter	Description
Input Voltage Range	0V to V_{ref} (typically 0 – 5V)
Output Digital Range	0 to 1023
Analog Pin Numbers	A0, A1, A2, A3, A4, A5
ADC Resolution	10-bit
Number of Levels	1024
Default Reference Voltage	5V (AVcc)

Important Functions

- ✍ `analogRead(pin)`: Reads the analog voltage present on a specified analog input pin and converts it into a 10-bit digital value (0-1023) using the internal ADC. The returned value represents the ratio of the input voltage to the selected reference voltage.
- ✍ `analogReference(type)`: Selects the reference voltage used by the ADC for analog-to-digital conversion. This reference determines the maximum measurable voltage and the resolution.
  - ☞ **DEFAULT**: Uses the Arduino supply voltage (typically 5V) as the reference, allowing the ADC to measure input voltages from 0V up to the supply voltage.
  - ☞ **INTERNAL**: Uses the internal 1.1 V bandgap reference of the ATmega328P, providing higher resolution for low-voltage measurements but with limited absolute accuracy.
  - ☞ **EXTERNAL**: Uses an externally applied reference voltage on the AREF pin (0-5V max), enabling precise control of ADC resolution when a stable reference is provided.

Table 6.2: Analog Read related Functions

Function Name	Purpose
`analogRead(pin)`	Reads analog voltage from the specified analog pin
`analogReference(type)`	Selects the ADC reference voltage

Reference Voltage

1. DEFAULT Reference (AVcc ≈ 5 V)

Selected using `analogReference(DEFAULT)`. The reference voltage is the Arduino supply voltage.

$$\text{Resolution} = \frac{V_{ref}}{1024} = \frac{5}{1024} \approx 4.88 \text{ mV/step}$$

Each increment in ADC output represents approximately 4.88 mV change in input voltage.

2. INTERNAL Reference (1.1 V)

Selected using `analogReference(INTERNAL)`. This uses the internal 1.1 V bandgap reference of the ATmega328P.

$$\text{Resolution} = \frac{V_{ref}}{1024} = \frac{1.1}{1024} \approx 1.07 \text{ mV/step}$$

This mode offers higher resolution for low-voltage measurements.

3. EXTERNAL Reference (AREF Pin)

Selected using `analogReference(EXTERNAL)`. An external reference voltage (0 – 5V max) is applied to the AREF pin.

$$\text{Resolution} = \frac{V_{ref(ext)}}{1024}$$

For example, if $V_{ref} = 3.3$ V:

$$\text{Resolution} = \frac{3.3}{1024} \approx 3.22 \text{ mV/step}$$

Table 6.3: Reference Voltage

Name	Voltage	Resolution
DEFAULT	5.0 V	4.88 mV
INTERNAL	1.1 V	1.07 mV
EXTERNAL	User defined (0 – 5V)	$V_{ref}/1024$

ADC Value Calculation

Now, the Arduino Uno features a 10-bit ADC. This means it maps input voltages between 0 and the reference voltage into integer values ranging from 0 to 1023 (which is $2^{10} - 1$).

The formula to calculate the digital output ADC value is:

$$\text{ADC} = \left(\frac{V_{\text{in}}}{V_{\text{ref}}}\right) \times 1023$$

Where:

- **ADC**: The digital number returned by `analogRead()`
- $V_{\textbf{in}}$: The actual analog voltage applied to the analog pin
- $V_{\textbf{ref}}$: The reference voltage the Arduino uses to compare against

Note: The ADC drops any decimals, so the final value is always rounded down to the nearest whole number (an integer).

Case 1: DEFAULT Reference Voltage (5V)

By default, the Arduino Uno uses its operating voltage of **5V** as the reference voltage ($V_{\text{ref}} = 5\text{V}$).

Scenario A: Input voltage is 2.5V (Half of V_{ref})

- $V_{\textbf{in}}$: 2.5V
- **Calculation**: ADC $= \left(\frac{2.5}{5}\right) \times 1023 = 0.5 \times 1023 = 511.5$
- **Arduino reads**: **511**

Scenario B: Input voltage is 1.2V

- $V_{\textbf{in}}$: 1.2V
- **Calculation**: ADC $= \left(\frac{1.2}{5}\right) \times 1023 = 0.24 \times 1023 = 245.52$
- **Arduino reads**: **245**

Case 2: INTERNAL Reference Voltage (1.1V)

The Arduino Uno has a built-in internal reference of **1.1V**. This means now $V_{\text{ref}} = 1.1\text{V}$ and 1.1V will read as 1023, and you cannot measure voltages higher than 1.1V.

Scenario: Input voltage is 0.8V

- $V_{\textbf{in}}$: 0.8V
- **Calculation**: ADC $= \left(\frac{0.8}{1.1}\right) \times 1023 = 0.727 \times 1023 = 743.99$
- **Arduino reads**: **743**

Case 3: EXTERNAL Reference Voltage (e.g., 3.3V)

You can supply your own clean reference voltage to the **AREF** pin. A common use case is connecting the Arduino's 3.3V pin to the AREF pin to read 3.3V sensors more accurately.

Scenario: Input voltage is 2.0V

- $V_{\mathbf{ref}}$: 3.3V
- $V_{\mathbf{in}}$: 2.0V
- **Calculation**: ADC $= \left(\frac{2.0}{3.3}\right) \times 1023 = 0.606 \times 1023 = 620.0$
- **Arduino reads**: **620**

Case 4: Input Voltage Exceeds Reference Voltage

This condition is known as **ADC saturation** or "clipping". Because the Arduino Uno has a 10-bit ADC, the absolute maximum digital value its hardware registers can hold is 1023. Even if the math suggests a higher number, the Arduino will cap the result at 1023.

Scenario: Input voltage is 4.5V, Reference is 3.3V – Let's say you have set the reference voltage to 3.3V using `analogReference(EXTERNAL)`, but you connect a sensor that outputs 4.5V to the analog pin.

- $V_{\mathbf{ref}}$: 3.3V
- $V_{\mathbf{in}}$: 4.5V
- **Theoretical Calculation**: ADC $= \left(\frac{4.5}{3.3}\right) \times 1023 = 1.363 \times 1023 = 1394.8$
- **Arduino reads**: **1023**

Why does this happen?

The formula calculates a value of roughly 1395. However, since the maximum possible output of a 10-bit ADC is exactly 1023 (represented in binary as `1111111111`), it cannot count any higher. Any voltage equal to or greater than the reference voltage will simply return 1023.

> **!** **Hardware Safety Warning:** While the ADC caps at 1023, you must never apply a voltage to an Arduino Uno pin that is significantly higher than its operating voltage (5V). Applying voltages greater than 5.5V can physically destroy the microcontroller, regardless of what the reference voltage is set to!

ADC Multiplexer Selection Register

Although the Arduino UNO uses several registers to control its Analog-to-Digital Converter (ADC), ADMUX (ADC Multiplexer Selection Register) deserves special attention because it directly governs how an analog signal is selected and interpreted. This register acts as a multiplexer controller, determining which analog input channel (A0–A5 on Arduino UNO) is connected to the ADC, while also configuring the reference voltage and result alignment. High-level functions such as `analogRead()` and `analogReference()` internally configure ADMUX before starting a conversion, meaning every analog measurement implicitly depends on this register. Understanding ADMUX therefore bridges the gap between Arduino's simple API and the underlying AVR hardware, enabling precise control, optimization, and effective troubleshooting of analog input behavior on the Arduino UNO. Now, ADMUX is a 8-bit register. The bit-level layout of the ADMUX register is shown below:

Bit	7	6	5	4	3	2	1	0
Name	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0

Reference Selection

The Reference Selection Bits, REFS1 and REFS0 (bits 7 and 6 of the ADMUX register), determine the voltage reference that the ADC uses for conversions. The reference voltage sets the maximum voltage that corresponds to the highest ADC value (1023) in a 10-bit conversion. On the Arduino UNO (ATmega328P), these two bits can be configured in four ways:

Table 6.4: Reference Selection Bits

REFS1	**REFS0**	**Reference Voltage**	**Command**
0	0	External reference voltage applied to the AREF pin. Must be supplied externally.	`analogReference(EXTERNAL)`
0	1	Uses the supply voltage (5V) as the reference. This is the default.	`analogReference(DEFAULT)`
1	0	Not recommended; behavior is undefined.	–
1	1	Uses the internal 1.1V reference voltage. Useful for measuring low voltages with higher precision.	`analogReference(INTERNAL)`

Resolution Selection

Bit 5 of the ADMUX register, called ADLAR (ADC Left Adjust Result), controls how the 10-bit ADC result is stored across the two 8-bit registers, ADCH (high byte) and ADCL (low byte). By default, the result is right-aligned, meaning the full 10-bit value occupies the lower 10 bits: the lower 8 bits go into ADCL and the upper 2 bits into ADCH. Setting ADLAR = 1 left-aligns the result, so the 10-bit value is shifted to the high bits of ADCH and ADCL, leaving the lower 6 bits of ADCL unused. This is particularly useful when only the high 8 bits are needed, allowing faster 8-bit precision ADC readings to be accessed directly from ADCH without additional bit manipulation. Left adjustment can improve efficiency in applications where full 10-bit resolution is not required.

Table 6.5: ADC Left Adjust Result Bit

ADLAR	Alignment	Resolution	DEFAULT
0	Right	10-bit	YES
1	Left	8-bit	–

Channel Selection

Bits 3 to 0 of the ADMUX register, labeled MUX3:MUX0, are used to select which analog input channel is connected to the ADC on the Arduino UNO. For standard Arduino UNO analog pins A0–A5, MUX3 is always 0, and MUX2:MUX0 determine the input pin number. The correct configuration of MUX bits is essential for `analogRead()` to read the intended analog signal.

Table 6.6: Analog Channel Selection Bits

MUX2	MUX1	MUX0	Pin/Source	Command
0	0	0	A0	`analogRead(A0)`
0	0	1	A1	`analogRead(A1)`
0	1	0	A2	`analogRead(A2)`
0	1	1	A3	`analogRead(A3)`
1	0	0	A4	`analogRead(A4)`
1	0	1	A5	`analogRead(A5)`

Analog Read Operation

Now, we are going to measure two different types of voltages. First, a fixed reference voltage is measured by connecting the Arduino's 3.3 V pin to an analog input pin A0 as shown in Figure 6.1. Next, the fixed connection is replaced with a potentiometer connected between 5V and GND, with the middle (wiper) terminal connected to the same analog pin as shown in Figure 6.2. Now, as the knob is rotated, the input voltage varies from 0V to 5V. We are going to use Code 6.1 for both of these setups.

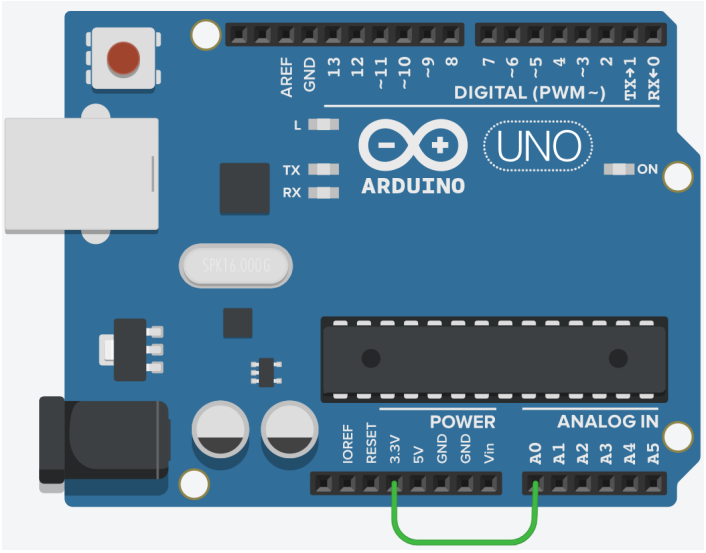


Figure 6.1: Reading Analog Voltage (3.3V)

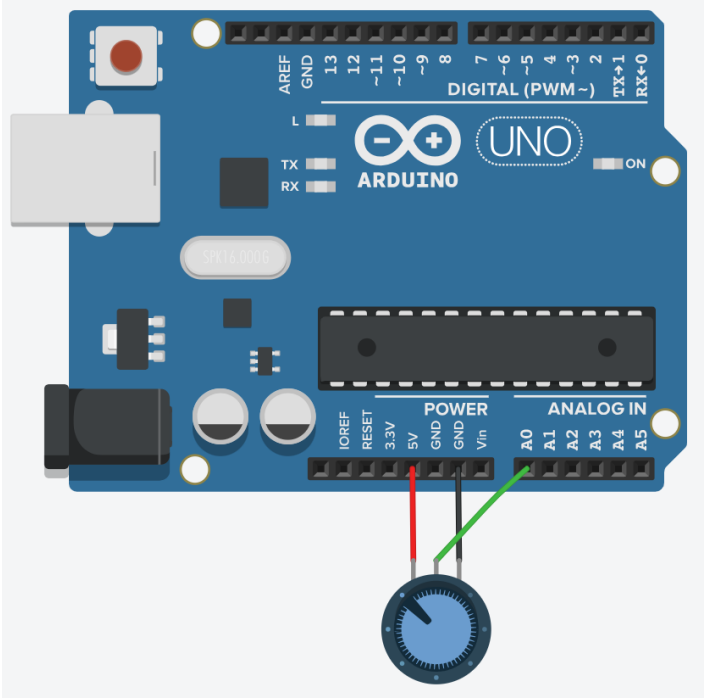


Figure 6.2: Potentiometer Voltage Reading

The following code uses the `analogRead()` function to convert the potentiometer's input voltage into a digital ADC value. It then determines the corresponding voltage by scaling this value with the reference voltage (5V) and ADC resolution (1023) using floating-point calculations.

$$V_{\text{in}} = \frac{\text{ADC} \times V_{\text{ref}}}{\text{Resolution}} = \frac{\text{ADC} \times 5}{1023}$$

Code 6.1 Analog Read

```
/*
    In this code we will learn the Basics of analogRead() function.

    We will connect the 3.3V pin or Potentiometer with any available Analog Pin.

    We will print both ADC and Analog Voltage value in Serial Monitor.
*/


#define sensorPin A0 // Analog Pin (A0) is Used for Analog Read Operation

// Defining Variables
int ADCvalue = 0; // 10-bit ADC Output (0 to 1023)
float Voltage = 0; // Analog Voltage

void setup()
{
  analogReference(DEFAULT); // Default 5V --> 1023 (ADC Value)
  Serial.begin(9600); // For Serial Monitor
  // ---- NO pinMode() Required ---- //
}

void loop()
{
  // Reading Analog Value
  ADCvalue = analogRead(sensorPin);

  // Floating-point operation using Type Casting
  Voltage = (float) ADCvalue * 5 / 1023; // Explicit
  // Voltage = ADCvalue * 5.0 / 1023;    // Implicit


  // Printing Value in Serial Monitor
  Serial.print("Voltage: ");
  Serial.print(Voltage);
  Serial.print("V");

  Serial.print('\t'); // tab (space)

  Serial.print("ADC: ");
  Serial.println(ADCvalue);

  delay(1000); // taking reading at 1sec interval
}
```

B Analog Write

Pulse Width Modulation (PWM) is a technique used to simulate an analog output using a digital signal. Instead of generating a continuously varying voltage, PWM rapidly switches a digital output between HIGH and LOW states. By controlling the proportion of time the signal remains HIGH within a fixed time period, known as the duty cycle, the average voltage delivered to a load can be adjusted. Due to this averaging effect, devices such as LEDs, motors, and buzzers respond as if they are receiving a true analog signal. PWM is widely used in microcontroller systems because it is efficient, flexible, and easily implemented using digital hardware timers. In Arduino, `analogWrite()` is used to generate an analog-like output on specific digital pins by producing a PWM signal. The following table summarizes the key characteristics of PWM signal generation. This information provides a concise reference for understanding how PWM signals are generated and controlled in the Arduino Uno.

Table 6.7: PWM in Arduino UNO

Parameter	**Description**
Output Type	Pulse Width Modulated (PWM) Digital Signal
Resolution	8-bit (256 discrete levels)
Value Range	0 to 255
PWM Pins	3, 5, 6, 9, 10, 11
Default PWM Frequency	$\sim$ 490 Hz (3, 9, 10, 11)
High PWM Frequency	$\sim$ 980 Hz (5 and 6)
Voltage Levels	0V (LOW) and 5V (HIGH)
Generation Method	Hardware timer based PWM
Control Method	Duty cycle variation
Duty Cycle Range	0% to 100%
Built-in Function	`analogWrite()`
Typical Applications	LED dimming, motor speed control

Functional Operation

- PWM generation is handled internally using hardware timers.
- A timer repeatedly counts from zero up to a predefined maximum value.
- At the start of each timer cycle, the output pin is set HIGH.
- When the timer count matches a compare value, the output pin is set LOW.
- The compare value determines how long the signal stays HIGH within one cycle.
- Changing this compare value alters the duty cycle of the PWM signal.

analogWrite()

- **Syntax**
  - `analogWrite(pin, value)`
- **Arguments**
  - `pin`
    - A digital pin that supports PWM output (Pin – 3, 5, 6, 9, 10, 11)
  - `value`
    - An integer between 0 and 255 (Determines the duty cycle of the PWM signal)

PWM Signal Parameters Calculation

The `analogWrite()` value is referred to as PWM duty cycle control value or simply PWM value.

The duty cycle is related to the PWM Value by the following formula:

$$\text{Duty Cycle (\%)} = \frac{\text{PWM Value}}{255} \times 100$$

Alternatively, we can calculate PWM value from duty cycle:

$$\text{PWM Value} = \frac{\text{Duty Cycle (\%)} \times 255}{100}$$

The average output voltage is:

$$V_{avg} = \text{Duty Cycle} \times V_{cc} \ [V_{cc} = 5V]$$

After establishing the mathematical relationship between the `analogWrite()` i.e. PWM control value, duty cycle, and the resulting average output voltage, it is useful to illustrate these relationships through several practical examples. The following examples demonstrate how different duty cycle percentages correspond to specific `analogWrite()` i.e. PWM control values and their associated average output voltages. By considering commonly used duty cycles such as 0%, 25%, 50%, 75%, and 100%, the relationship between the PWM control value and the effective output voltage can be more clearly understood.

- **0% Duty Cycle**

$$\text{PWM Value} = 0$$
$$\text{Duty Cycle} = \frac{0}{255} \times 100 = 0\%$$
$$V_{\text{avg}} = 0 \times 5 = 0\,\text{V}$$

- **25% Duty Cycle**

$$\text{PWM Value} \approx 64$$
$$\text{Duty Cycle} = \frac{64}{255} \times 100 \approx 25\%$$
$$V_{\text{avg}} = 0.25 \times 5 = 1.25\,\text{V}$$

- **50% Duty Cycle**

$$\text{PWM Value} \approx 127$$
$$\text{Duty Cycle} = \frac{127}{255} \times 100 \approx 50\%$$
$$V_{\text{avg}} = 0.5 \times 5 = 2.5\,\text{V}$$

- **75% Duty Cycle**

$$\text{PWM Value} \approx 191$$
$$\text{Duty Cycle} = \frac{191}{255} \times 100 \approx 75\%$$
$$V_{\text{avg}} = 0.75 \times 5 = 3.75\,\text{V}$$

- **100% Duty Cycle**

$$\text{PWM Value} = 255$$
$$\text{Duty Cycle} = \frac{255}{255} \times 100 = 100\%$$
$$V_{\text{avg}} = 1 \times 5 = 5\,\text{V}$$

The relationship between the `analogWrite()` value and the resulting PWM signal is illustrated in the following figure and table. The figure shows how different `analogWrite()` values correspond to different duty cycles of the PWM waveform. As the value provided to `analogWrite()` i.e. PWM control value increases from 0 to 255, the duty cycle of the signal gradually increases, meaning the output remains HIGH for a longer portion of each cycle.

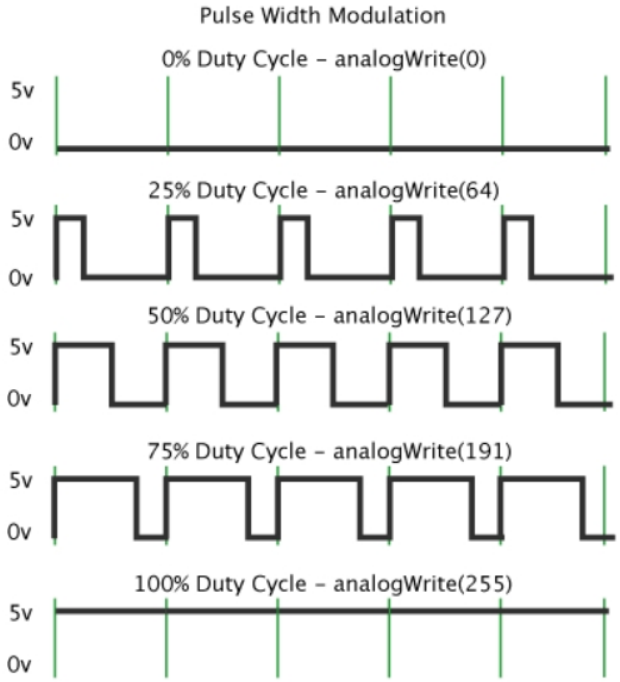


Figure 6.3: PWM & Duty Cycle

The accompanying table summarizes several representative `analogWrite()` values along with their corresponding duty cycles and average output voltages. This helps demonstrate how PWM can be used to simulate different analog voltage levels using a digital output pin.

Table 6.8: PWM Duty Cycle

Duty Cycle (%)	**PWM Value**	**Average Voltage**
0	0	0.00 V
25	64	1.25 V
50	127	2.50 V
75	191	3.75 V
100	255	5.00 V

Analog Write Operation

For Analog Write, we can use any PWM (∼) pin. Here, we have used Digital Pin 9. A LED with current limiting resistor has been added to that pin as shown in Figure 6.4. Corresponding breadboard connection for this setup is shown in Figure 6.5. Both Code 6.2 & Code 6.3 demonstrates this Analog Write process using this same setup.

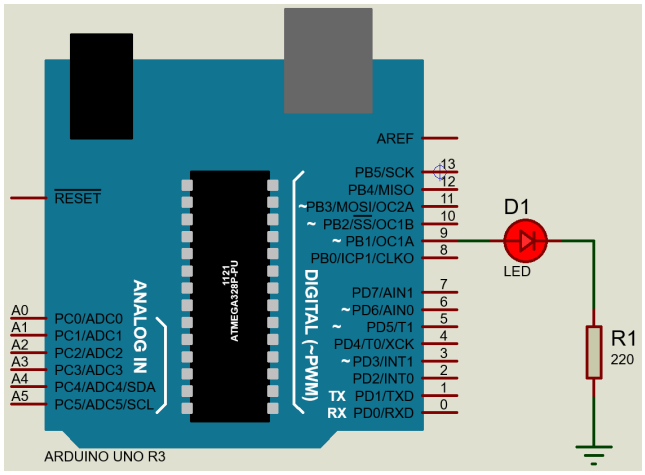


Figure 6.4: Analog Write (PWM) Operation

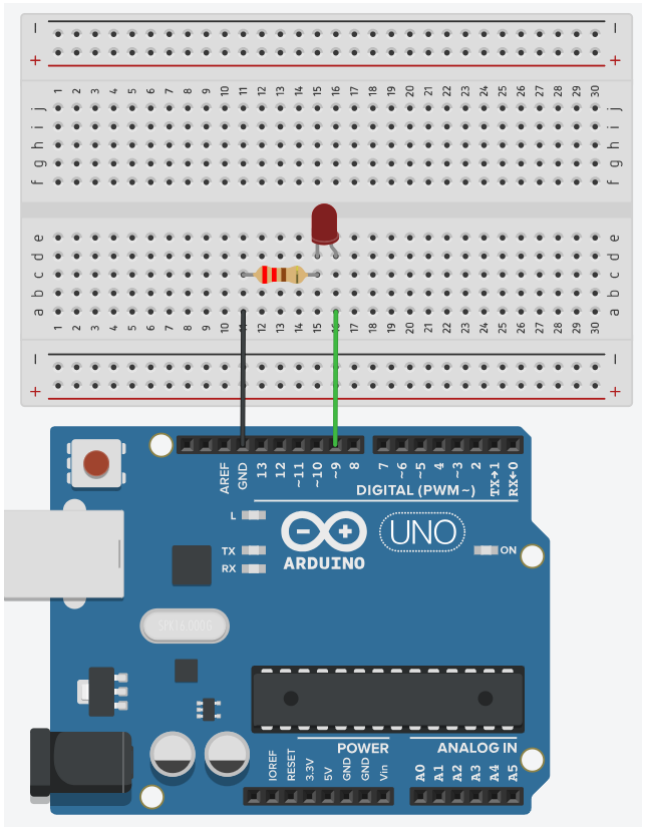


Figure 6.5: Breadboard Connection of Figure 6.4

.

The following code reads a PWM value entered by the user through the Serial Monitor and uses it to control the brightness of an LED. It first prompts the user to enter a PWM value, then displays the received input value. To prevent invalid or out-of-range values, the program applies the `constrain()` function to limit the input within the valid 8-bit PWM range of 0–255, ensuring reliable and safe operation of the output device. This step avoids unexpected behavior that could occur if the user enters values beyond the allowable range. Finally, the `analogWrite()` function generates a PWM signal corresponding to the constrained value, where the duty cycle of the signal determines the LED's brightness — lower values produce dim light and higher values produce greater intensity.

To learn more about the `constrain()` function, refer to Code 1.17 on Page 35.

Code 6.2 Analog Write

```
/*
    In this code we will learn the Basics of analogWrite(). We will take a number as
    a input from user and create the Pulse Wave Modulation (PWM) signal. As the PWM
    signal has 8-bit value, we will constrain the user input from 0 to 255. It will
    ultimately control the brightness of connected LED to that particular pin.

    PWM pins are identified with   a "~" sign, like ~3, ~5, ~6, ~9, ~10 and ~11.
*/


// LED connection with Arduino
#define LED 9

int PWMvalue = 0; // 8-bit Pulse Width Modulation (PWM) Control Value

void setup()
{
  Serial.begin(9600);
  // pinMode NOT required
}

void loop()
{
  // Taking Input From User
  Serial.print("PWM: ");
  while (Serial.available() == 0) {}
  PWMvalue = Serial.parseInt();
  Serial.println(PWMvalue);

  // Constraining Value in 8-bit range [0-255]
  PWMvalue = constrain(PWMvalue, 0, 255);

  // PWM Signal
  analogWrite(LED, PWMvalue);
}
```

The following code gradually varies the brightness of an LED by generating a pulse-width modulation (PWM) signal on an Arduino board. The LED brightness is controlled using an 8-bit PWM value ranging from 0 to 255, while the delay interval in milliseconds between brightness changes determines how fast the brightness varies. Inside the `loop()` function, the first `for` loop progressively increases the brightness by incrementing PWM value in steps of 5, producing a smooth fade-in effect as the LED intensity rises from minimum to maximum. After reaching full brightness, the second `for` loop gradually decreases the PWM value in the same step size, creating a fade-out effect. The `analogWrite()` function generates the corresponding PWM signal for each value, and the `delay()` function ensures a short pause between successive changes, resulting in a continuous LED fading pattern.

Code 6.3 Fading LED

```
/*
  Fade
  This example shows how to fade an LED using the analogWrite() function.
  We will connect a LED to a Pulse Width Modulation (PWM) enabled pin.

  PWM pins are identified with  a "~" sign, like ~3, ~5, ~6, ~9, ~10 and ~11.
*/


// LED connection with Arduino
#define LED 9


// Defining Variables
int PWMvalue = 0; // Brightness [8-bit PWM Value (0-255)]
int dTime = 30; // Delay Time (ms) Milli Seconds


void setup()
{
  // pinMode NOT required
}

void loop()
{
  // Increasing Brightness
  for (PWMvalue = 0; PWMvalue <= 255; PWMvalue += 5)
  {
    analogWrite(LED, PWMvalue);
    delay(dTime); // Wait for dTime millisecond(s)
  }

  // Decreasing Brightness
  for (PWMvalue = 255; PWMvalue >= 0; PWMvalue -= 5)
  {
    analogWrite(LED, PWMvalue);
    delay(dTime); // Wait for dTime millisecond(s)
  }
}
```

C Analog Read & Analog Write

Here, a potentiometer is used to control the brightness of an LED by using both `analogRead()` and `analogWrite()` in the same program. The middle terminal of the potentiometer is connected to the analog input pin A0 to read a variable voltage between 0V and 5V, while the LED is connected to the PWM output pin 9. As the knob is rotated, `analogRead()` converts the changing voltage into a digital ADC value, which is then mapped to a PWM duty cycle and sent to the LED using `analogWrite()`. This causes the LED brightness to increase or decrease smoothly according to the position of the potentiometer. Required breadboard connection for this setup is shown in Figure 6.6 and the corresponding program is Code 6.4.

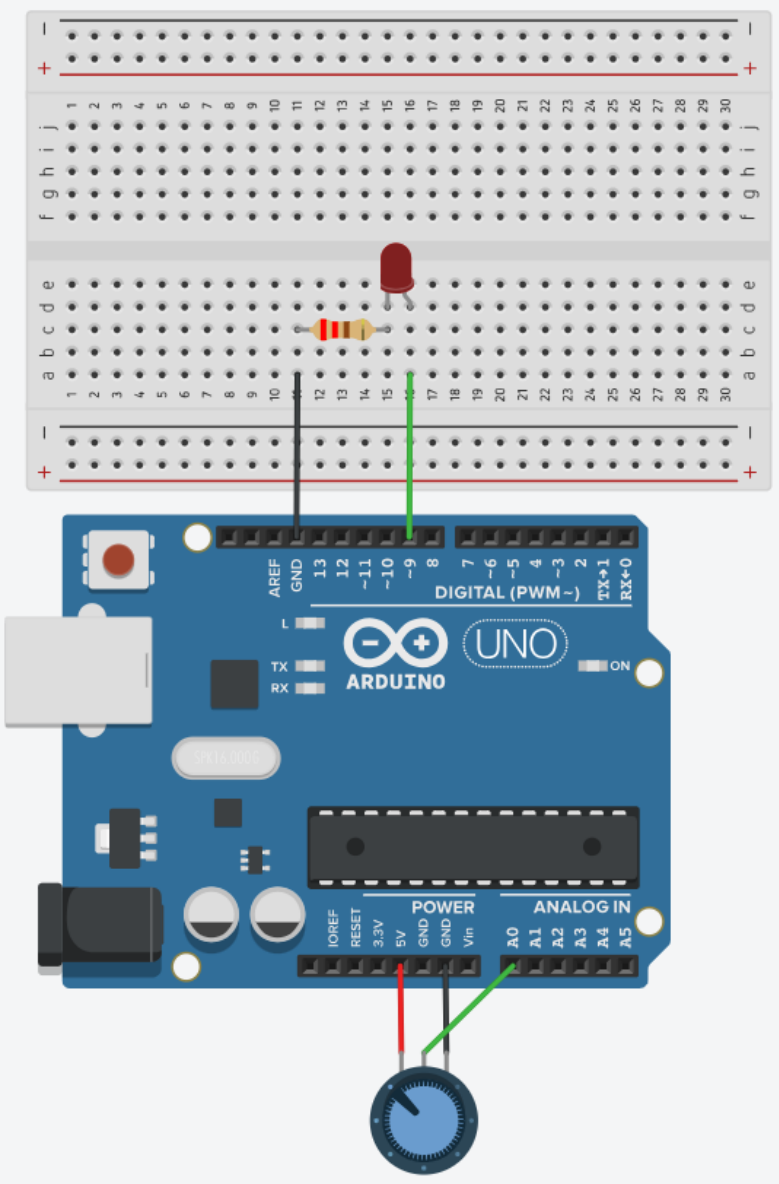


Figure 6.6: Dimmable LED Control using Potentiometer

The following code shows how to control the brightness of an LED using a potentiometer by reading an analog input and converting it into a Pulse Width Modulation (PWM) output. Here, LED is connected to a pin which supports PWM. The `analogRead()` function measures the voltage from the potentiometer, returning a 10-bit value ranging from 0 to 1023. Because the `analogWrite()` function relies on an 8-bit PWM signal, this reading must be scaled down to a range of 0 to 255, which is handled seamlessly by the `map()` function. Finally, the program outputs this mapped value to the LED to adjust its brightness and pauses execution for some milliseconds using `delay()` function to give the analog-to-digital converter (ADC) a moment to stabilize before taking the next reading.

To learn more about the `map()` function, refer to Code 1.17 on Page 35.

Code 6.4 Analog Read & Write

```
/*
    In this code we would change the brightness of LED based on the Potentiometer
    Voltage Reading. We are going to use analogRead() & analogWrite() function.
    We will use the map() function to set the Pulse Width Modulation (PWM).
*/


// Potentiometer & LED connection with Arduino
#define sensorPin A0
#define LED 9


// Defining Variables
int ADCvalue = 0; // Analog Read
int PWMvalue = 0; // Analog Write


void setup()
{
  // pinMode NOT required
}

void loop()
{
  // Reading the Analog Voltage from ADC
  ADCvalue = analogRead(sensorPin);

  // Mapping the Value
  PWMvalue = map(ADCvalue, 0, 1023, 0, 255);

  // PWM Signal
  analogWrite(LED, PWMvalue);

  // wait for ADC converter to settle
  delay(10);
}
```

Chapter 7

Keypad

A membrane keypad is a low-profile human–machine interface device widely used in embedded systems and electronic instruments for user input. It consists of multiple flexible layers - typically a top graphic overlay, spacer, and conductive circuit layer - laminated together to form a compact, sealed structure. When a key is pressed, the top layer flexes and makes contact with the conductive traces below, completing an electrical circuit that can be detected by a microcontroller. Due to their thin form factor, low cost, light weight, and resistance to dust and moisture, membrane keypads are commonly found in applications such as calculators, microwave ovens, access control panels, and educational development boards. The sealed construction also improves durability in harsh or frequently used environments compared to mechanical switches.

4×4 Membrane Keypad

A 4×4 membrane keypad is one of the most widely used configurations, offering 16 keys arranged in four rows and four columns. The keys are internally connected in a matrix format, which requires only 8 microcontroller I/O pins (4 row lines and 4 column lines) instead of 16 individual connections (Figure 7.1), making the interface highly efficient and hardware-friendly. When a key is pressed, it electrically connects a specific row and column, and the microcontroller identifies the key using a row–column scanning technique by sequentially driving rows and monitoring column states. To ensure stable logic levels during scanning, internal or external pull-up resistors are commonly used. These keypads typically operate at standard logic voltages such as 5V, ensuring compatibility with popular platforms like Arduino microcontrollers. The keys are usually labeled with numeric digits (0–9), arithmetic symbols, and function keys such as A–D or *, #, making the 4×4 membrane keypad well suited for applications like password-based access control systems, menu navigation, digital calculators, and other simple embedded user interfaces.

Internal Structure

Inside a 4×4 membrane keypad, the switches are arranged in a matrix of 4 rows and 4 columns, forming 16 switch points at their intersections. Each key is located at the crossing of one specific row line and one specific column line. Internally, all keys in the same row share a common horizontal conductor, and all keys in the same column share a common vertical conductor.

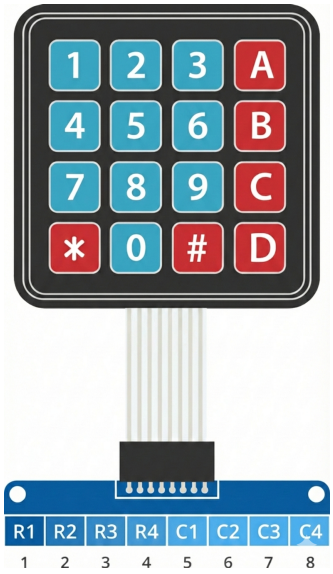


Figure 7.1: 4×4 Keypad Pins

When a key is pressed, it electrically connects its corresponding row and column, creating a unique row–column pair. The controller identifies the pressed key by driving rows or columns one at a time and reading the others, a method known as matrix scanning. This structure allows 16 keys to be read using only 8 signal lines instead of 16 individual connections.

Scanning Procedure

In row–column scanning, the microcontroller does not check all keys at once; instead, it activates one row at a time and checks which column becomes connected. First, all column pins are configured as inputs, usually with pull-up resistors enabled, and all row pins are configured as outputs, with all rows initially kept at logic HIGH. The controller then sets the first row to logic LOW while keeping the other rows HIGH; if any key in that row is pressed, it electrically connects that LOW row to its corresponding column, causing that column input to read LOW. By checking which column went LOW, the controller knows exactly which key in that active row was pressed, and the combination of the active row number and detected column number uniquely identifies the key. After reading the columns, the controller sets the first row back to HIGH and pulls the second row LOW, repeating the same checking process for all rows one by one. This scanning is done continuously in a loop and very quickly, typically many times per second, so the user experiences it as instant detection; if no column changes during a particular row scan, it means no key in that row is pressed. Only one row is activated at a time to avoid confusion between multiple keys, and software usually adds a small debouncing delay of a few milliseconds to prevent false multiple detections caused by mechanical vibration of the key contacts.

Keypad Connection with Arduino

When connecting a keypad to Arduino, digital pins 0 and 1 are kept unused because they are reserved for serial communication (RX and TX) with the computer, so that serial communication through the USB port works properly for uploading programs and displaying keypad output on the Serial Monitor. Typically, the four row wires of the keypad are connected to digital pins 9,

8, 7, and 6, while the four column wires are connected to digital pins 5, 4, 3, and 2, though any available digital pins can be used. The detailed pin connections used in this setup are listed in the following table, and the corresponding circuit diagram is illustrated in the following figure.

Table 7.1: Keypad Connection with Arduino

Rows	Arduino	Columns	Arduino
R1	9	C1	5
R2	8	C2	4
R3	7	C3	3
R4	6	C4	2

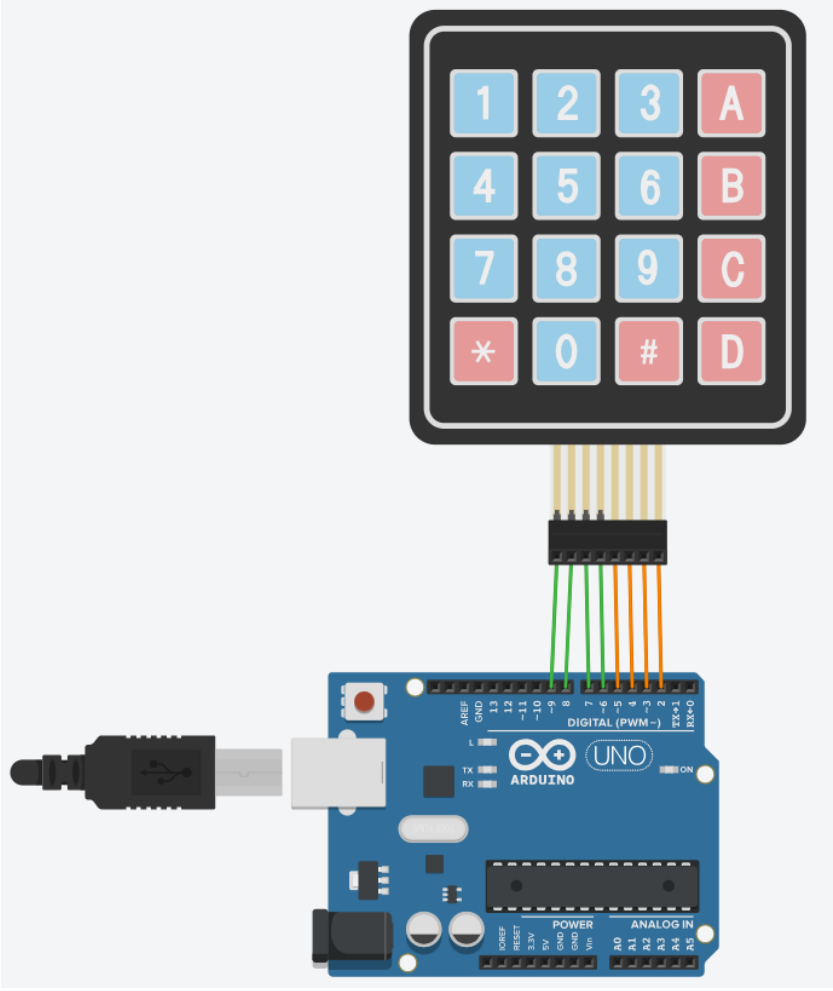


Figure 7.2: Keypad Connection with Arduino

ASCII Encoding

American Standard Code for Information Interchange (ASCII) is a character encoding system used to represent text in computers and digital devices by assigning a unique numeric value to letters, digits, punctuation marks, and control characters. Since computers operate using binary (0s and 1s) and cannot directly understand letters or symbols, numbers are used to represent characters so that textual information can be stored, processed, and transmitted electronically. Standard ASCII uses 7 bits to represent 128 different characters. By providing a standardized numeric code for each character, ASCII enabled different machines, computers and communication systems to communicate consistently.

Table 7.2: Keypad Characters with ASCII Values

Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII
#	35	2	50	6	54	A	65
*	42	3	51	7	55	B	66
0	48	4	52	8	56	C	67
1	49	5	53	9	57	D	68

NULL Character

The null character (`\0`) is a special character which represents an empty value or the absence of data. It has an ASCII value of 0 and does not produce any visible output when printed. The null character indicates that no valid input or character is present, such as when a function returns no data. In the Keypad library, it is represented by `NO_KEY`, which means no key is pressed.

Methods

In the Keypad library, there are two main methods available to detect key presses. The `getKey()` method is non-blocking and simply checks if a key has been pressed, returning the key value or `NO_KEY` (`\0`) if nothing is pressed. On the other hand, `waitForKey()` is blocking and pauses the program until a key is pressed, ensuring that input is received before the program continues.

Table 7.3: Keypad Methods

Method	Type	Function
`getKey()`	Non-blocking	Returns pressed key or NULL if no key is pressed
`waitForKey()`	Blocking	Waits untill a key is pressed and returns the key

Basic Operation

This program demonstrates the basic operation of a matrix keypad interfaced with an Arduino. The code continuously scans the keypad to detect key presses and displays the corresponding character on the serial monitor.

Code 7.1 Basic Keypad

```
/*
  This is the most simple and introductory code for a 4*4 Keypad.
  In this code we will print the pressed key in the serial monitor.
*/


// Custom Library Inclusion
#include <Keypad.h>


// Specifying no. of rows & columns of keypad
const int ROW_NUM = 4; // four rows
const int COLUMN_NUM = 4; // four columns


// Character Map
char keys[ROW_NUM][COLUMN_NUM] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};


// Keypad connection with Arduino Pin
byte pin_rows[ROW_NUM] = {9, 8, 7, 6}; // Rows (R1,R2,R3,R4) connection to Arduino
byte pin_column[COLUMN_NUM] = {5, 4, 3, 2}; // Columns (C1,C2,C3,C4) connection to Arduino


// Declaring a object under Keypad Class library
Keypad mykeypad = Keypad( makeKeymap(keys), pin_rows, pin_column, ROW_NUM, COLUMN_NUM );


void setup() {
  Serial.begin(9600); // Serial monitor to print characters
}


void loop() {
  char key = mykeypad.getKey(); // getkey() method to get the key

  // Checking whether any key has been pressed to avoid Printing Null Characters
  // ----- key ** key == true ** key != false ** key != 0 ** key != '\0' ----- //
  if (key != NO_KEY) {
    Serial.println(key);
  }
}
```

Numeric Keypad

This program demonstrates a modified keypad interface where only numeric inputs (0–9) are accepted and displayed, while all other keys such as A, B, C, D, * and # are ignored. This selective input approach is useful in applications where only numerical data is required, such as password entry or number-based user input. This implementation shows how input validation can be performed in a Arduino keypad system to restrict unwanted key presses. The program continuously checks the pressed key and verifies whether it falls within the acceptable numeric range before displaying or storing it. This method enhances system security, reduces input errors, and simplifies further data processing by ensuring that the collected input follows a predefined format. Such controlled input handling is commonly used in embedded systems that require accurate and restricted data entry, and it demonstrates an important technique for designing robust and user-friendly keypad interfaces.

Code 7.2 Numeric Keypad

```
/*
  Here we would create a simple Number Pad. That means it would only take input when a
  user presses any number [0 - 9]. If user presses any letter ('A','B','C','D') or any
  special character ('*','#') it would simply ignore those key inputs.
*/

#include <Keypad.h>

const int ROW_NUM = 4; //four rows
const int COLUMN_NUM = 4; //four columns

char keys[ROW_NUM][COLUMN_NUM] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};

byte pin_rows[ROW_NUM] = {9, 8, 7, 6}; //connect to the row pinouts of the keypad
byte pin_column[COLUMN_NUM] = {5, 4, 3, 2}; //connect to the column pinouts of the keypad

Keypad mykeypad = Keypad( makeKeymap(keys), pin_rows, pin_column, ROW_NUM, COLUMN_NUM );

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

void loop() {
  char key = mykeypad.getKey();

  // Checking whether any number key has been pressed (Excluding letters)
  if (key >= '0' && key <= '9') {
    Serial.println(key);
  }
}
```

Alphabetic Keypad

This program also demonstrates a modified keypad interface where alphabetic inputs (A–D) and special characters such as * and # are accepted and processed, while only numeric keys (0–9) are ignored. This approach is useful in applications where only letter-based input is required, such as mode selection or command entry. This implementation also illustrates how input filtering can be applied in a Arduino keypad system to ensure that only relevant key presses are recognized. The program continuously monitors key presses and checks whether the detected input belongs to the allowed set of characters before processing or displaying it. This controlled input handling improves system efficiency, reduces errors, and simplifies program logic by eliminating unnecessary data validation at later stages. Such techniques are widely used in embedded systems that require structured command input, demonstrating an important method for designing reliable and application-specific keypad interfaces.

Code 7.3 Alphabetic Keypad

```
/*
  Here we would create a simple Number Pad. That means it would only take input when a
  user presses any letter ('A',B','C','D') or any special character ('*','#'). If user
  presses any number ['0' - '9'] it would simply ignore those key inputs.
*/

#include <Keypad.h>

const int ROW_NUM = 4; //four rows
const int COLUMN_NUM = 4; //four columns

char keys[ROW_NUM][COLUMN_NUM] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};

byte pin_rows[ROW_NUM] = {9, 8, 7, 6}; //connect to the row pinouts of the keypad
byte pin_column[COLUMN_NUM] = {5, 4, 3, 2}; //connect to the column pinouts of the keypad

Keypad keypad = Keypad( makeKeymap(keys), pin_rows, pin_column, ROW_NUM, COLUMN_NUM );

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

void loop() {
  char key = keypad.getKey();

  // Checking whether any letter key has been pressed (Excluding Numbers)
  if (key != NO_KEY && key < '0' || key > '9') {
    Serial.println(key);
  }
}
```

Simple Calculator Design

This program implements a simple keypad-based calculator that performs arithmetic operations on two single-digit numbers, including addition (+), subtraction (–), multiplication (*), division (/), power (∧), and remainder (%). The keypad mapping is modified so that specific keys represent these operations: pressing 'A' produces '+', pressing 'B' produces '–', pressing 'C' produces '*', pressing 'D' produces '/', pressing '*' produces '∧' and pressing '#' produces '%'. This demonstrates how the keypad layout can be customized to perform specific functions, along with basic input processing and arithmetic computation using a Arduino keypad interface.

A ⇒ Prints **'+'** and performs the **addition** operation.

B ⇒ Prints **'−'** and performs the **subtraction** operation.

C ⇒ Prints **'*'** and performs the **multiplication** operation.

D ⇒ Prints **'/'** and performs the **division** operation.

***** ⇒ Prints **'∧'** and performs the **power** operation.

⇒ Prints **'%'** and performs the **remainder** operation.

Now while checking the operation key, a validity condition has been included to ensure that only the designated operator keys are accepted; if a numeric key is pressed at this stage, the program identifies it as an invalid input and displays an appropriate message. However, for simplicity, no such validation has been implemented during the number input stage. As a result, if the user presses non-numeric character keys available on the keypad while entering the digits, these inputs are still processed by the program and may lead to unintended or incorrect results.

Code 7.4 Simple Calculator </>

```
/*
    This code shows how to construct a simple calculator which takes two single-digit
    input from keypad and performs various mathematical operation between them.
*/

#include <Keypad.h>

const int ROW_NUM = 4; //four rows
const int COLUMN_NUM = 4; //four columns

char keys[ROW_NUM][COLUMN_NUM] = {
  {'1', '2', '3', '+'}, // A (Addition)
  {'4', '5', '6', '-'}, // B (Subtraction)
  {'7', '8', '9', '*'}, // C (Multiply)
  {'^', '0', '%', '/'}  // * (Power --> ^) + # (Remainder --> %) + D (Division --> /)
};

byte pin_rows[ROW_NUM] = {9, 8, 7, 6}; //connect to the row pinouts of the keypad
byte pin_column[COLUMN_NUM] = {5, 4, 3, 2}; //connect to the column pinouts of the keypad

Keypad myKeypad = Keypad( makeKeymap(keys), pin_rows, pin_column, ROW_NUM, COLUMN_NUM );
```

```
void setup() {

  Serial.begin(9600); // Initializing Serial Monitor
}


void loop() {

  // Taking First Input
  Serial.print("First Number: ");
  int n1 = myKeypad.waitForKey() - '0'; // ASCII to Integer Conversion
  Serial.println(n1);

  // Operation Input
  Serial.print("Operation: ");
  char op = myKeypad.waitForKey(); // Mathematical Operator
  Serial.println(op);

  // Taking Second Input
  Serial.print("Second Number: ");
  int n2 = myKeypad.waitForKey() - '0'; // ASCII to Integer Conversion
  Serial.println(n2);

  // Calculation Result
  int result = calculator(n1, op, n2); // Calling Custom Function
  Serial.print("Result = ");
  Serial.println(result);
  Serial.println("------------------------"); // Creating a space between calculations
}



// Custom Function for Mathematical Calculation
int calculator (int n1, char op, int n2) {

  int result = 0; // Variable for storing calculation result

  if (op == '+')
    result = n1 + n2;
  else if (op == '-')
    result = n1 - n2;
  else if (op == '*')
    result = n1 * n2;
  else if (op == '/')
    result = n1 / n2;
  else if (op == '%')
    result = n1 % n2;
  else if (op == '^')
    result = round(pow(n1, n2));
  else
    Serial.println("Invalid Operation");

  return result;
}
```

Multiple Digit Input

When interfacing a 4×4 keypad with an Arduino Uno, multiple digit input enables the system to receive a sequence of key presses rather than a single character. Instead of processing each key independently, the microcontroller collects and stores successive inputs to form a complete numeric value or command. This feature is essential in embedded applications such as password authentication, electronic locks, calculator inputs, security systems, and menu-driven interfaces, where accurate and organized data entry is required for reliable system operation.

To handle such input efficiently, two main approaches are commonly used:

- Fixed Length Digit Input
- Variable Length Digit Input

In the fixed length approach, the number of digits to be entered is predetermined by the program, and the system processes the input only after receiving the required number of keys. In contrast, variable length input allows users to enter digits freely and uses a specific termination key to indicate the end of the sequence. The choice between these methods depends on system requirements, flexibility, and application design.

Fixed Length Digit Input

Fixed length digit input refers to a method where the system is programmed to accept a specific number of key presses, such as a 3-digit or 4-digit code. The microcontroller reads each key press sequentially and stores the input in a buffer or variable until the required number of digits is reached. After receiving the complete sequence, the system performs further actions such as verification, comparison with a stored value, or execution of a command. This approach is simple to implement and ensures consistent input size, reducing the need for additional checks or termination conditions. It is widely used in applications like door security systems, PIN-based authentication, and access control mechanisms, where the input format is fixed and known in advance. The fixed-digit method which requires a predetermined number of key presses before an operation is executed demonstrated in Code 7.5.

Variable Length Digit Input

Variable length digit input allows the user to enter an arbitrary number of digits without a predefined limit. In this method, the system continuously monitors keypad presses and stores each input until a designated terminating key, such as '#', is pressed to signal the completion of data entry. Once the termination key is detected, the stored sequence is processed by the program. This technique provides greater flexibility compared to fixed length input, as it supports different input sizes and user-defined data lengths. It is commonly used in applications such as numeric data entry, menu selection systems, and command-based interfaces, where the amount of input may vary depending on user requirements. The variable digit method which uses a special terminating key to gather inputs of flexible length, allowing numbers of any size demonstrated in Code 7.6. In this implementation, additional input validation has also been incorporated to verify whether a numeric key has been pressed before storing the data. Non-numeric keys are ignored during number collection, ensuring that only valid digits contribute to the final value.

Code 7.5 Fixed Length Number

```
/*
    This code shows how to take a fixed digit input from a keypad
    and then multiplies the given input by 2 and show the result.
    First, it prompts the user to specify the input length
    and then takes that exact number of digits from user.
*/

#include <Keypad.h>

const int ROW_NUM = 4; //four rows
const int COLUMN_NUM = 4; //four columns

char keys[ROW_NUM][COLUMN_NUM] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};

byte pin_rows[ROW_NUM] = {9, 8, 7, 6}; //connect to the row pinouts of the keypad
byte pin_column[COLUMN_NUM] = {5, 4, 3, 2}; //connect to the column pinouts of the keypad

Keypad mykeypad = Keypad( makeKeymap(keys), pin_rows, pin_column, ROW_NUM, COLUMN_NUM );

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

void loop() {

  // Taking no. of digits as Input (Value must be greater than Zero)
  Serial.print("Number of Digits = ");
  int n = mykeypad.waitForKey() - '0'; // ASCII to Integer Conversion
  Serial.println(n);


  // Taking Multi Digit Input
  Serial.print("Input Number: ");

  int keyval = 0; // Keypad value input
  int value = 0; // variable for storing input (accumulator)

  for (int i = 1; i <= n; i ++)
  {
    keyval = mykeypad.waitForKey() - '0'; // ASCII to Integer Conversion
    Serial.print(keyval); // Printing Digits one by one
    value = value * 10 + keyval; // Multi Digit Conversion
  }

  Serial.println(); // Printing new line to indicate end of input
  Serial.print("Result = ");
  Serial.println(value * 2); // User input multiplied by 2
}
```

Code 7.6 Variable Length Number

```
/*
    This code shows how to take a Integer input from user using String Class object.
    After taking the input it shows the result which is the user input multiplied by 2.
    It checks input validity and user has to press "#" to denote the end of the input.
*/

#include <Keypad.h>

const int ROW_NUM = 4; //four rows
const int COLUMN_NUM = 4; //four columns

char keys[ROW_NUM][COLUMN_NUM] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};

byte pin_rows[ROW_NUM] = {9, 8, 7, 6}; //connect to the row pinouts of the keypad
byte pin_column[COLUMN_NUM] = {5, 4, 3, 2}; //connect to the column pinouts of the keypad

Keypad mykeypad = Keypad( makeKeymap(keys), pin_rows, pin_column, ROW_NUM, COLUMN_NUM );

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

void loop() {

  char key = 0;      // Pressed key
  int num = 0;       // Converted Integer Input Number

  // String Input (Initialized as an Empty String)
  String str = "";  // ** NO Space between double quotes **

  Serial.print("Input: "); // Taking Multi Digit Input

  while (key != '#')  // Denotes End of Input String
  {
    key = mykeypad.waitForKey();  // Fetching new keypress until '#' is pressed

    if (key >= '0' && key <= '9')  // Checking Valid key Input
    {
      Serial.print(key); // To print individual valid keystrokes only
      str += key;        // Concatenating pressed keys
    }
  }

  num = str.toInt(); // String to Integer conversion
  Serial.println();  // Printing new line to indicate end of input
  Serial.print("Answer: ");
  Serial.println(num * 2); // User input multiplied by 2
}
```

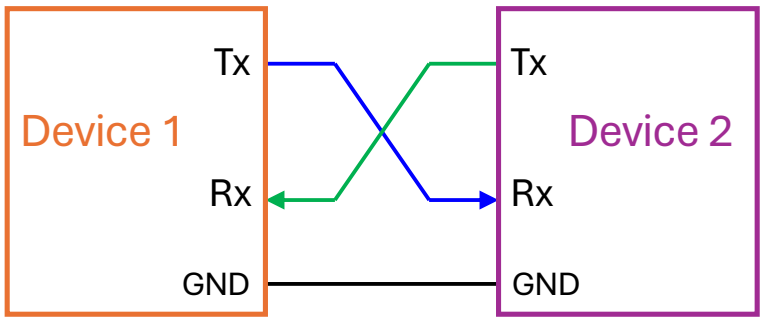
Chapter

8 SUART Communication

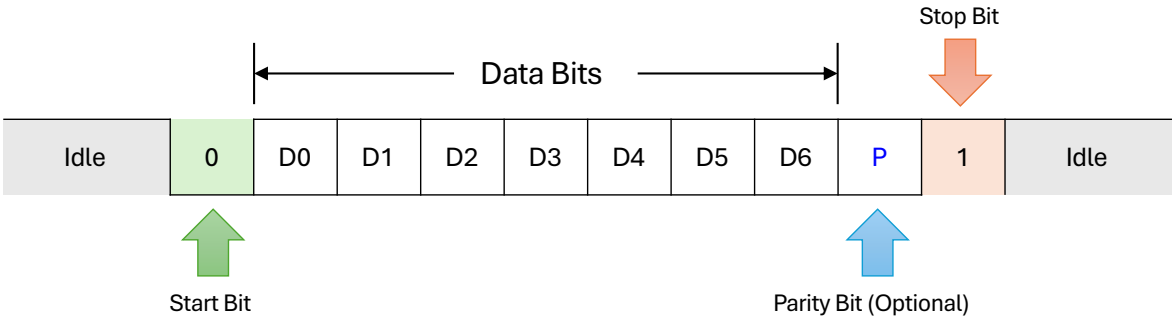
The Arduino Uno contains a single hardware Universal Asynchronous Receiver Transmitter (UART) interface on digital pins RX (D0) and TX (D1), which is internally connected to the USB-to-serial converter and therefore becomes unavailable for general-purpose communication whenever the board is programmed or the serial monitor is used. In many embedded applications, multiple serial peripherals such as Bluetooth modules, GPS receivers, GSM modems, wireless transceivers, and serial sensors must be interfaced simultaneously, which motivates the use of a software-based UART implementation using ordinary digital input/output pins.

UART Protocol

Software UART operates using a technique known as bit-banging, where the microcontroller manually controls and samples the logic level of a pin in software while maintaining strict timing accuracy to emulate asynchronous serial communication. In this approach, serial data is transmitted and received without dedicated hardware shift registers or baud rate generators; instead, the processor executes delay loops or timer-based routines to generate precise bit intervals according to the selected baud rate. A typical UART frame begins with an idle high line condition, followed by a start bit represented by a logic low level, then a sequence of data bits transmitted least significant bit first, optional parity for error detection, and one or more stop bits at logic high, with the most common configuration being 8 data bits, no parity, and one stop bit (8N1). The bit duration is inversely proportional to the baud rate and is given by $T_{bit} = 1/\text{Baud}$, such that for a baud rate of 9600 bps the bit period is approximately $104\,\mu$s, requiring the microcontroller to maintain microsecond-level timing accuracy throughout the entire frame transmission and reception process. During transmission, the software UART drives the transmit pin low for one bit time to generate the start bit, sequentially outputs each data bit with carefully timed delays, and finally restores the pin to the idle high state for the stop bit, thereby reconstructing a complete asynchronous serial frame entirely in software. During reception, the input pin continuously monitors for a falling edge indicating the start bit, waits approximately half a bit time to align sampling at the center of the first data bit, and then samples the logic level at fixed intervals equal to the bit duration in order to reconstruct the received byte, followed by validation of the stop bit to confirm frame integrity.



(a) Connection



(b) Frame

Figure 8.1: UART Protocol

Library

This functionality is most commonly provided by the `SoftwareSerial` library on the Arduino Uno platform, which allows users to define custom receive and transmit pins and supports standard serial functions such as `begin()`, `read()`, `write()`, and `available()`, thereby providing an interface similar to the built-in hardware serial port. Internally, the library uses pin change interrupts to detect the start bit on the receive pin and disables interrupts for carefully controlled intervals during transmission to maintain accurate timing, which directly affects system responsiveness and limits multitasking capability.

Limitation

Because all timing is managed by software, the achievable baud rate is constrained by processor speed, interrupt latency, and the execution time of background tasks; in practice, reliable communication on the Arduino Uno is typically achieved in the range of 300 bps to 38400 bps, while higher baud rates such as 57600 bps or 115200 bps may lead to framing errors, data corruption, or missed characters, especially when other interrupt-driven peripherals such as timers, pulse-width modulation outputs, or sensor libraries are active. Furthermore, software UART generally operates in a half-duplex manner during reception because the processor must dedicate significant execution time to sampling the incoming signal, and only one `SoftwareSerial` instance can actively listen at any given moment, which limits scalability in multi-device systems. Electrical reliability also depends on stable logic levels, minimal noise, and short wire lengths, since asynchronous serial communication lacks inherent clock synchronization and relies entirely on accurate timing alignment between transmitter and receiver. Despite these limitations, software

UART provides substantial flexibility by allowing serial communication on almost any available digital pin, conserving the hardware UART for debugging or USB communication and enabling rapid prototyping without additional hardware.

Communication Modes

Serial communication operation mode can be divided into two categories depending on the direction of data flow between communicating devices –

- One Way Communication
- Two Way Communication

One Way Communication

One-way communication, also known as simplex communication, allows data transmission in only a single direction from a transmitter to a receiver, such as when a sensor sends continuous measurement data to a controller without receiving any response or control commands. This mode is simple to implement, requires minimal synchronization, and reduces processing overhead, making it suitable for monitoring, broadcasting, and data logging applications where feedback is unnecessary.

Two Way Communication

Two-way communication enables bidirectional data exchange between devices and may operate in half-duplex or full-duplex modes. In half-duplex communication, both devices can transmit and receive data but not simultaneously, requiring turn-taking and collision avoidance mechanisms, whereas in full-duplex communication, data can be transmitted and received at the same time using separate transmit and receive channels. In the context of software-based UART on microcontrollers such as the Arduino Uno, communication is typically half-duplex due to timing constraints and processor limitations, while hardware UART supports full-duplex operation. Two-way communication is essential for command-response systems, configuration interfaces, and real-time control applications where acknowledgments, error handling, and interactive data exchange are required.

Hardware Connection

Two Arduino Uno boards needs to be interfaced for Software UART based communication. Since Software UART allows serial communication through user-defined digital pins, any suitable digital I/O pins can be selected instead of the default hardware TX (pin 1) and RX (pin 0). In this configuration, digital pin 2 of the first Arduino Uno is connected to digital pin 3 of the second Arduino Uno, and digital pin 3 of the first board is connected to digital pin 2 of the second board, forming a proper cross-connection between transmit (TX) and receive (RX) lines. This cross-wiring ensures that the transmitting pin of one board is linked to the receiving pin of the other. Additionally, the GND pins of both boards are connected together to establish a common ground reference, which is necessary for stable and accurate serial data exchange between the two microcontrollers. The connection configuration for the Software UART interface between the two boards is presented in Table 8.1 and the corresponding wiring diagram illustrating these connections is shown in Figure 8.2 providing a visual representation of the TX–RX cross-connection and the common ground linkage required for proper serial communication.

Table 8.1: SUART Connection

Uno1	Uno2
2 (Rx Pin)	3 (Tx Pin)
3 (Tx Pin)	2 (Rx Pin)
GND	GND

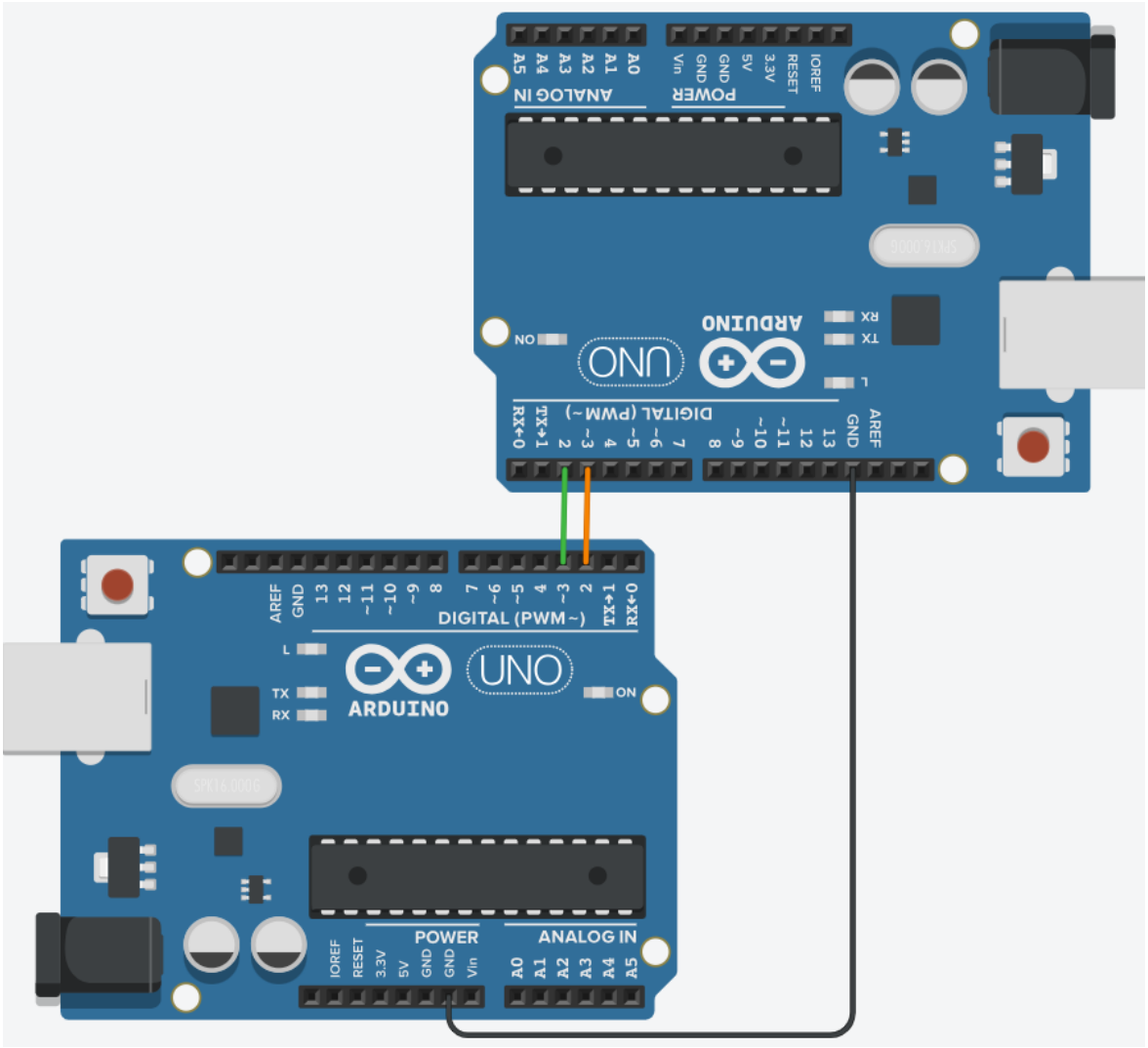


Figure 8.2: SUART Connection between Two Arduino UNOs

A One Way Communication

In one-way (simplex) communication, data flows in a single direction - from the transmitter to the receiver. The transmitter program is shown in Code 8.1T, while the corresponding receiver program is shown in Code 8.1R. The transmitter sends the data, and the receiver only receives it, with no feedback or return communication. In the transmitter code, an integer value entered by the user is transmitted to the receiver.

Code 8.1T One Way Communication (Transmitter UNO)

```
/*
   This Code shows how to Transmit (send) data over SUART.

   First, it will take input (Integer Number) from user
   and then send this data using SUART to other UNO device.

   It will also print the data in its serial monitor.
*/


// Including Library
#include <SoftwareSerial.h>


// Defining Pins for Receive & Transmit
#define RxPin 2
#define TxPin 3


// Defining SUART Ports for Transmitter UNO
SoftwareSerial SUART_UNOTx(RxPin, TxPin);


void setup() {
  Serial.begin(9600); // Initializing Serial Monitor
  SUART_UNOTx.begin(9600); // Initializing SUART Communication
}

void loop() {

  // Taking Input from User
  Serial.print("Input = ");
  while (Serial.available() == 0) {}
  int a = Serial.parseInt();

  // Printing in Serial Monitor
  Serial.println(a);

  // Sending Data through SUART to Receiver UNO
  SUART_UNOTx.print(a); // Do NOT write print*ln* <-- Extra (\n + \r)
}
```

In one-way (simplex) communication, the receiver does not send any response back; its role is limited to receiving and processing the incoming data. After the data is received, the receiver processes it to generate a suitable output for the intended application. In this experiment, the receiver reads the integer value sent by the transmitter and performs a simple mathematical operation by squaring the received input before displaying the result. Receiver program is shown in the following code.

Code 8.1R One Way Communication (Receiver UNO)

```
/*
   This Code will show how to receive data sent through SUART.

   First Receiving UNO will receive the transmitted data.
   Then it will squared the received integer number.

   Then it will print the squared value of the
   received number in its Serial Monitor.
*/


// Including the Library
#include <SoftwareSerial.h>


// Defining Pins for Receive & Transmit
#define RxPin 2
#define TxPin 3


// Declaring SUART Object for Receiver UNO
SoftwareSerial SUART_UNORx(RxPin, TxPin);


void setup() {
  Serial.begin(9600); // Initializing Its Serial Monitor
  SUART_UNORx.begin(9600); // Initialing SUART Communication
}


void loop() {

  // Checking whether data is available in its SUART Port
  if (SUART_UNORx.available()) {
    int b = SUART_UNORx.parseInt(); // Saving Data Received through SUART
    Serial.print("Squared Term = ");
    Serial.println(b * b); // Printing Squared Value
  }

}
```

B Two Way Communication

In two-way (duplex) communication, there is no dedicated transmitter or receiver. Instead, both devices act as transceivers, meaning each can transmit and receive data. In this setup, communication occurs in both directions. As a result, almost the same program is written for both transceivers, with only minor changes depending on the specific role at a given time. Here, We will design a chat box between two devices where String messages will be exchanged and shown like traditional Instant Messaging App services like "Whats-App" or "Messenger".

Code 8.2T Two Way Communication (UNO 1)

```
/*
  This Code is written for User Number 1.
*/


// Including Library
#include <SoftwareSerial.h>

// SUART Object Declaration for UNO1
SoftwareSerial SUART_UNO1(2, 3); //(RxPin,TxPin)

// ************ Change Your Name Here ************ //
String yName = "Your_Name"; // Put Your Name here

// String Variable for Message
String msgTx, msgRx; // Transmitted & Received message

void setup() {
  Serial.begin(9600); // Initializing Serial Monitor
  SUART_UNO1.begin(9600); // Initializing SUART communication
}

void loop() {

  // Transmitting Message to UNO2
  if (Serial.available())
  {
    msgTx = yName + ": " + Serial.readString(); // Adding Name (ID) before Message
    Serial.println(msgTx); // Printing Transmitted Message in its Serial Monitor
    SUART_UNO1.print(msgTx); // Transmitting (sending) message through SUART
  }

  // Receving Message from UNO2
  if (SUART_UNO1.available())
  {
    msgRx = SUART_UNO1.readString(); // Receiving message through SUART
    Serial.println(msgRx); // Printing Received message
  }

}
```

Code 8.2T and Code 8.2R are almost identical. The only difference between the two programs is the user-defined names used in the code; otherwise, the logic and functionality remain the same for both transceivers. For this SUART messaging system, sender identification is added to each transmitted message to make the communication clearer and more organized. Before sending the actual text, a predefined sender name or ID is appended at the beginning of the message. As a result, when the message is received and displayed, it includes the sender information followed by the message content. This approach works similarly to traditional messaging applications where each message is associated with a specific user. Including the sender ID helps distinguish messages from different devices and makes the conversation easier to understand.

Code 8.2R Two Way Communication (UNO 2)

```
/*
    This is same code written for User Number 2.
*/


// Including Library
#include <SoftwareSerial.h>

// SUART Object Declaration for UNO2
SoftwareSerial SUART_UNO2(2, 3); // (RxPin, TxPin)

// ************ Change Your Name Here ************ //
String yName = "Your_Name"; // Put Your Name here

// String Variable for Message
String msgTx, msgRx; // Transmitted & Received message

void setup() {
  Serial.begin(9600);
  SUART_UNO2.begin(9600);
}

void loop() {

  // Transmitting Message to UNO1
  if (Serial.available())
  {
    msgTx = yName + ": " + Serial.readString(); // Adding Name (ID) before Message
    Serial.println(msgTx); // Printing Transmitted (sent) message
    SUART_UNO2.print(msgTx); // Transmitting (sending) message through SUART
  }

  // Receving Message from UNO1
  if (SUART_UNO2.available())
  {
    msgRx = SUART_UNO2.readString(); // Receiving message sent through SUART
    Serial.println(msgRx); // Printing Received message
  }

}
```

C Device Control

In the next step, we develop a program in which a command is sent from the transmitter to control a device connected at the receiving end. This type of control can be implemented using either one-way or two-way communication; however, in this example, one-way communication is used. The transmitter sends a command to turn the device ON or OFF, and the receiver reads this command and controls the LED connected to it accordingly. The transmitter program is shown in Code 8.3T, while the corresponding receiver program is shown in Code 8.3R.

Code 8.3T Controlling Device (Transmitter UNO)

```
/*
   This Code shows how to Transmit (send) data over SUART.

   First, it will take input (String) from user
   and then send this data using SUART to other UNO device.

   It will also print the data in its serial monitor.
*/


// Including Library
#include <SoftwareSerial.h>

// Defining SUART Ports for Transmitter UNO
SoftwareSerial SUART_UNOTx(2, 3); // (RxPin, TxPin)

void setup() {
  Serial.begin(9600); // Initializing Serial Monitor
  SUART_UNOTx.begin(9600); // Initializing SUART Communication
}

void loop() {
  // Taking Input from User
  Serial.print("Command: ");
  while (Serial.available() == 0) {}
  String msgT = Serial.readString(); // Command (Message) to be Transmitted

  // Printing in Serial Monitor
  Serial.println(msgT);

  // Sending Data through SUART to Receiver UNO
  SUART_UNOTx.print(msgT); // Do NOT write print*ln* <-- Error
}
```

The following code shows the receiver program, which receives commands from the transmitter and simply executes them by controlling the connected device accordingly. After executing the action, the receiver displays a status message on its serial monitor indicating whether the LED was turned on, turned off, or if an invalid command was received. In this implementation, the communication is one-way, as commands are sent only from the transmitter to the

receiver. However, if the receiver sends these status messages back to the transmitter so they can also be displayed on the transmitter's serial monitor, the system would operate as a two-way communication setup.

Code 8.3R Controlling Device (Receiver UNO)

```
/*
   This Code will show how to receive data sent through SUART.

   First Receiving UNO will receive the transmitted data.

   If the received command (String):
      - "ON" than it will turn on the Built-in LED.
      - "OFF" than it will turn off the Built-in LED.
      -  For any other it will print "Invalid Command".
*/


// Including the Library
#include <SoftwareSerial.h>

// Declaring SUART Object for Receiver UNO
SoftwareSerial SUART_UNORx(2, 3); // (RxPin, TxPin)

void setup() {
  Serial.begin(9600); // Initializing Its Serial Monitor
  SUART_UNORx.begin(9600); // Initialing SUART Communication
  pinMode(LED_BUILTIN, OUTPUT); // Setting Pin Mode (Pin 13)
}

void loop() {
  if (SUART_UNORx.available()) // Checking whether data is avaialble in its SUART Port
  {
    String msgR = SUART_UNORx.readString(); // Received Command (Message) through SUART
    msgR.toUpperCase(); // Making Case Insensitive

    // Controlling LED depending on Received Data
    if (msgR == "ON")
    {
      digitalWrite(LED_BUILTIN, HIGH); // Turn "ON" LED
      Serial.println("LED turned ON");
    }
    else if (msgR == "OFF")
    {
      digitalWrite(LED_BUILTIN, LOW); // Turn "OFF" LED
      Serial.println("LED turned OFF");
    }
    else
    {
      Serial.println("Invalid Command");
    }
  }
}
```