People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University of Science and Technology of Oran - Mohamed BOUDIAF
Faculty of Electrical Engineering
Vice-Dean, Postgraduate, Scientific Research and External Relations
وزارة التعليم العالي والبحث العلمي
جامعة وهران للعلوم والتكنولوجيا محمد بوضياف
كلية الهندسة الكهربائية
نيابة عمادة الكلية لما بعد التدرج والبحث العلمي والعلاقات الخارجية

Département d'Electronique

Polycopié Pédagogique

Titre

Étude et Réalisation des Projets

Cours destiné aux étudiants de :

Licence (spécialité et niveau) : ……

Master (spécialité et niveau) : Electronique des Systèmes Embarqués – Master 2

Doctorat – D1 (spécialité : ……

Fait par : Mr DAOUD A.

Année Universitaire : 2025/2026

People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University of Science and Technology of Oran - Mohamed BOUDIAF
Faculty of Electrical Engineering
Vice-Dean, Postgraduate, Scientific Research and External Relations
وزارة التعليم العالي والبحث العلمي
جامعة وهران للعلوم والتكنولوجيا محمد بوضياف
كلية الهندسة الكهربائية
نيابة عمادة الكلية لما بعد التدرج والبحث العلمي والعلاقات الخارجية

Department of Electronics

Course Manual

Title

Project Design and Implementation

Course Manual for Second-Year Master's Students in:

Embedded Systems Electronics

Mr DAOUD A.

Academic Year: 2025/2026

Preface

This course manual has been developed as a comprehensive academic resource for students in the Master's program in "Electronique des Systèmes Embarqués / Étude et Réalisation des Projets," as well as educators and practitioners seeking a deeper understanding of the ATmega328P microcontroller within the Arduino ecosystem. The content is designed to support both independent study and classroom instruction. Each chapter builds upon foundational knowledge, integrating key programming concepts, hardware interfacing, sensor applications, timing mechanisms, and peripheral control. Hands-on examples and clear explanations are provided to reinforce core principles and encourage practical exploration.

Table of Contents

Introduction

Microcontroller-based platforms are now vital tools for engineering research, education, and real-world system development due to the rapid growth of embedded systems. Among these platforms, the Arduino ecosystem, particularly those built around the ATmega328P microcontroller, has become a widely adopted foundation for learning digital electronics, programming, and hardware–software integration. Notably, its open-source design, library support, and development environment are very conducive to education on basic concepts for embedded systems, while also facilitating in-depth studies of its internal operation.

This manual describes, in a structured and comprehensive way, the ATmega328P microcontroller implemented in Arduino boards such as the Uno and Nano. Students will learn how to design and build functional embedded systems by mastering increasingly complex theoretical concepts, practical programming techniques, hardware interfacing, and project-oriented applications. Thus, Chapter 1 provides a brief review of essential concepts and introduces the Microchip ATmega328P microcontroller used in both the Arduino Uno R3 and Nano boards. Chapter 2 presents an introduction to C programming for beginners. Chapters 3–5 offer a detailed exploration of the main features of the ATmega328P microcontroller, including analog-to-digital conversion, shift registers, port expanders, interrupts, serial communication interfaces, and timing mechanisms.

It is noticed that all sketches and code included in this manual have been compiled using Arduino IDE version 1.8.19 on a Windows 64-bit platform, simulated and verified in Proteus ISIS 8.17, and practically validated using an Arduino Uno R3 board, a breadboard, jumper wires, a digital oscilloscope, and other essential laboratory tools.

1.1 Introduction

Arduino is an open-source set of electronic microcontroller boards that may easily be programmed to understand and interact with their environment. It began as an open-hardware teaching/prototyping initiative and evolved into a family of AVR-based boards widely used in education, research, prototyping, and low-cost industry systems. Key tradeoffs are ease-of-use and ecosystem versus limited processing/real-time performance and constrained peripheral resources.

Arduino boards come in numerous types and forms, each designed for different use cases such as simple hobby projects, IoT, robotics, wearables, etc. Among them is the popular development board Uno R3 which is designed for electronics and coding, as well as for education. As shown in Fig.1.1, the Uno board is based on the ATmega328P and has digital and analog input/output pins, a 16 MHz crystal, a USB connector, a power jack, an ICSP header, and a reset button [1][2]. Therefore, it contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with an AC/DC adapter or battery to get started.

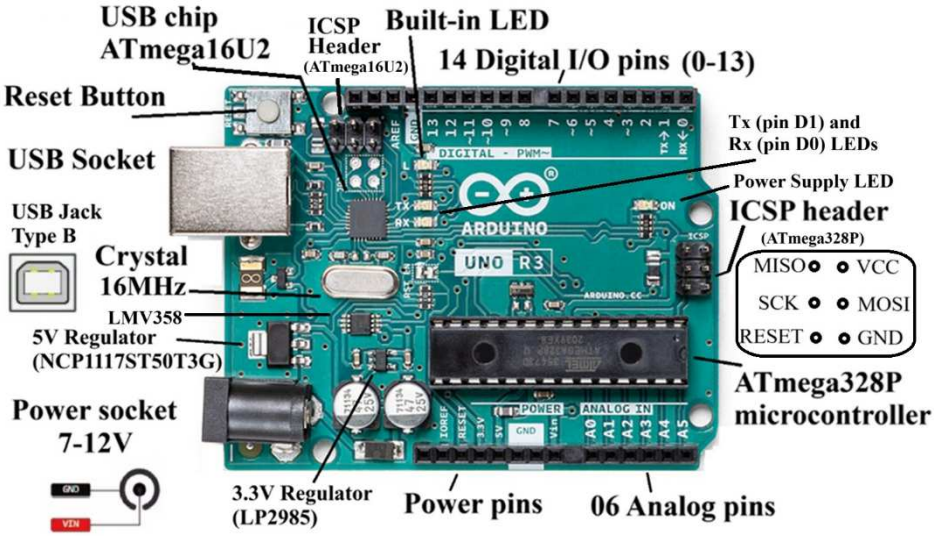


Fig.1.1: The Arduino Uno board

1.2 Arduino Uno Board

Technical specifications of the Uno board are summarized in the below table [1]:

Table 1.1: Technical specification

Feature	Specification
Microcontroller	ATmega328P (8-bit)
Operating Voltage	5 V
Input Voltage (recommended)	7 – 12 V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3 V Pin	50 mA
Flash Memory	32 KB (0.5 KB used by bootloader)
SRAM	2 KB

EEPROM	1 KB
Clock Speed	16 MHz
In-Circuit Serial Programmer Header (ICSP)	2 (for programming/debugging the ATmega328P and ATmega16U2)
Built-in LED Pin	1 (pin PB5 (D13))

The following diagram of Fig.1.2 is a pinout diagram for an Arduino Uno board based on the ATmega328P PDIP microcontroller [3]. It maps the functions of the pins on the board and uses a color-coded legend to categorize the different types of functionalities available on each pin. Furthermore, breakdown of the legend categories and their corresponding colors and functions are described in table 1.2.

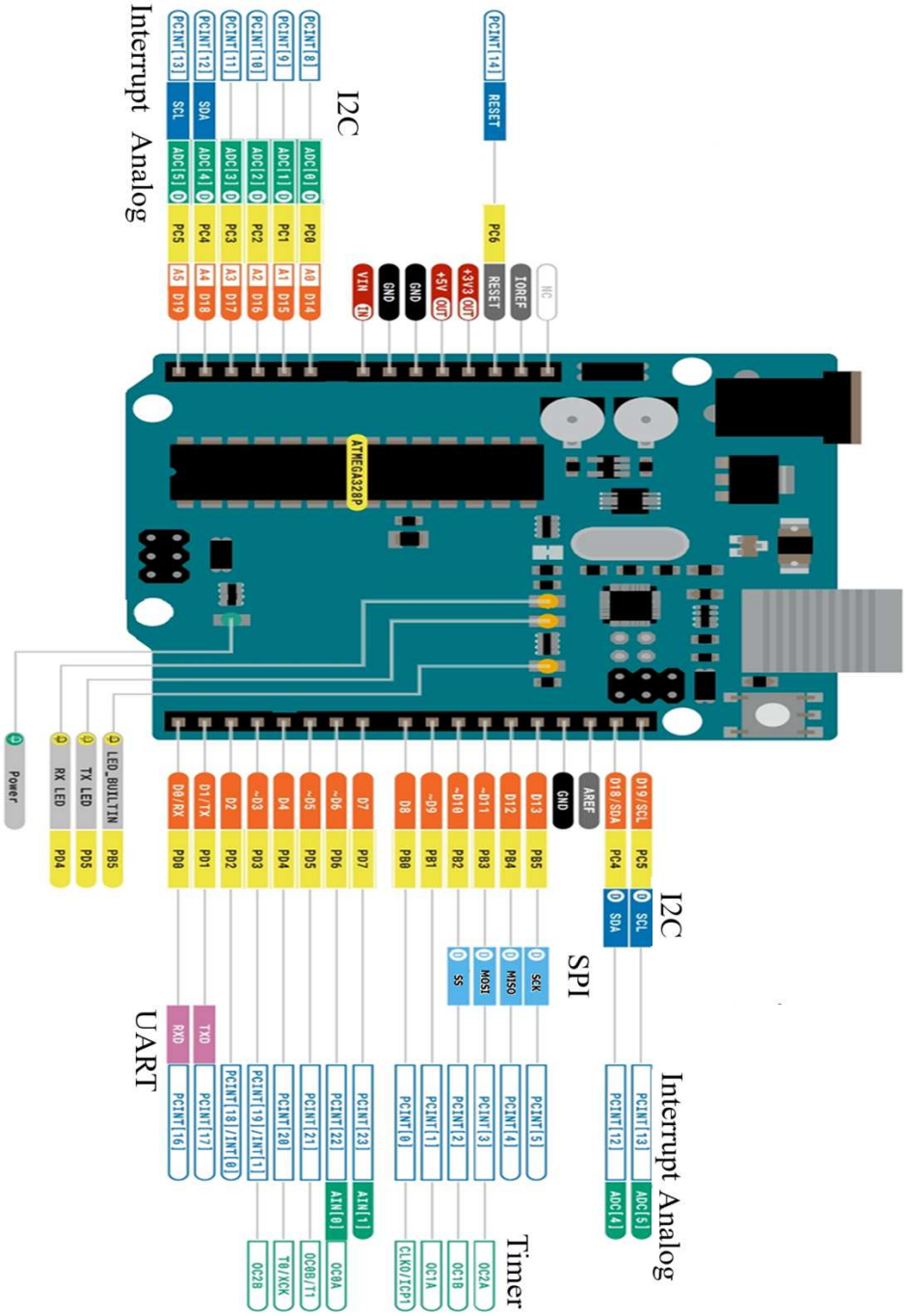


Fig.1.2: The Arduino Uno board full pinout

Table 1.2: Arduino Uno Pin Mapping and Roles

Function Category	Description
Power	3.3V: Provides 3.3V (max 50 mA) 5V: Regulated 5V supply from USB or regulator VIN: External input power (6–20V, recommended 7–12V) RESET: Resets the ATmega328P (active LOW), also accessible via PC6 IOREF: Outputs the board's operating voltage (5V) and provides a reference for shields. AREF: Analog reference pin for analog inputs (Default: 5V, 1.1V(Internal), ≤5V (External)).
Analog	A0–A5 (10-bit ADC), can also act as digital I/O
PWM / Timer	PWM-enabled pins: D3, D5, D6, D9, D10, D11
UART	D0 (RX), D1 (TX) for hardware serial
SPI	D10 (SS), D11 (MOSI), D12 (MISO), D13 (SCK)
I^2C	A4 (SDA), A5 (SCL)
Digital I/O	General-purpose: D0–D13
Interrupts	External: D2 (INT0), D3 (INT1)
Micro (Port mapping)	ATmega328P ports: PC0…PC5 = A0–A5, PD0…PD7 = D0 to D7, PB0...PB5 = D8 toD13
LED Indicators	LED_BUILTIN (D13), TX LED, RX LED
Ground	GND pins (0V reference)

There are multiple ways to power the Arduino Uno, such as:

- via USB connector
- via the onboard barrel jack connector
- via the VIN (Voltage In) pin
- via the 3.3V/5V pin

The Arduino UNO R3 connects to the host laptop or PC via a USB cable (Type A male to Type B female). It may be powered from the USB port during project development. However, it is highly recommended that an external power supply be employed. This will allow developing projects beyond the limited electrical current capability of the USB port. Also, Arduino (www.arduino.cc) recommends a power supply from 7–12 VDC with a 2.1-mm center positive plug. The UNO has two onboard voltage regulators that maintain the incoming power supply voltage to a stable 5 V and 3.3V.

1.3 Arduino Nano Board

Arduino Nano is an intelligent development board designed for building faster prototypes with the smallest dimension (Fig.1.3). At the heart of the board is ATmega328P microcontroller clocked at a frequency of 16 MHz featuring more or less the same functionalities as the Arduino Uno. Arduino Nano provides enough interfaces for the breadboard-friendly applications by offering 20 digital input/output pins, 8 analog pins, and a mini-USB port (Fig.1.4) [4].

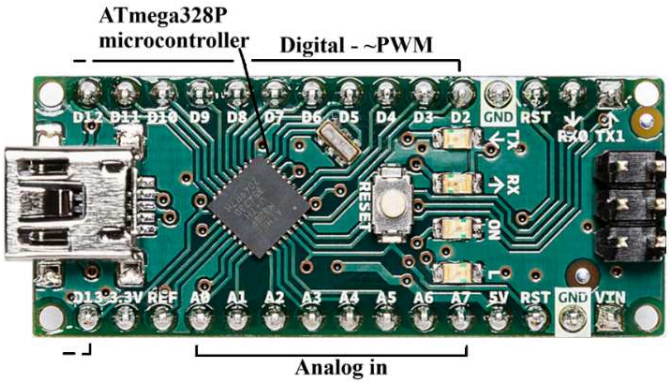


Fig.1.3: The official Arduino Nano board

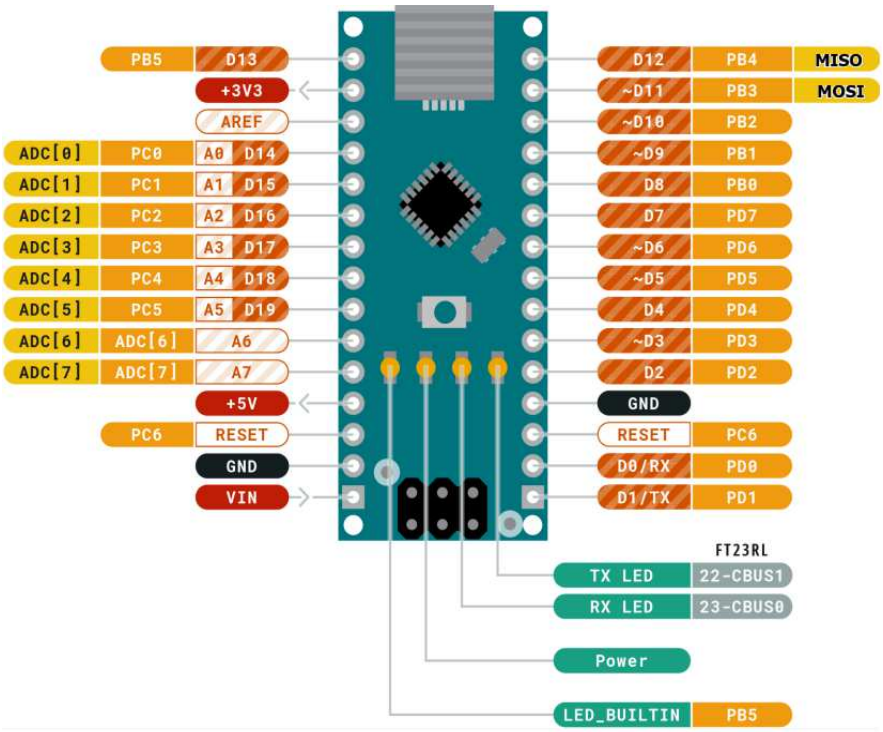


Fig.1.4: Nano board full pin-out

The Arduino Uno and Arduino Nano share the same ATmega328P microcontroller and have very similar functionality, but they differ mainly in size, connectivity, form factor, and the microcontroller package (PDIP (28-pin) on the Uno and TQFP (32-pin) on the Nano). The table below summarizes the main differences between the two boards.

Table 1.3: Arduino Uno vs. Arduino Nano

Feature	**Arduino Uno**	**Arduino Nano**
Analog Inputs	6 (A0–A5)	8 (A0–A7; A6 & A7 are analog-only)
Power Connector	DC barrel jack , VIN	VIN only
USB Connector	USB Type-B	Mini-USB or Micro-USB
USB-to-Serial Chip	ATmega16U2 (official boards)	FT232 / CH340 (varies by version)
Size / Form Factor	Larger (68.6 × 53.4 mm), shield-friendly	Smaller (45 × 18 mm), breadboard-friendly

1.4 Overview of Arduino Boards

The following table provides a comparison of various Arduino boards, excluding the Uno and Nano families. It highlights their key characteristics such as microcontroller type, flash memory size, operating voltage, I/O pins, and clock speed [5][6].

Table 1.4: Comparison of Common Arduino Boards

Board	Microcontroller	Operating Voltage	Flash Memory	Clock Speed	Digital I/O Pins
Arduino Mega 2560	ATmega2560, 8-bit	5V	256 kB 8kB bootloader	16 MHz	54
Arduino Micro	ATmega32U4, 8-bit	5V	32 kB	16 MHz	20
Arduino Leonardo	ATmega32U4	5V	32 kB	16 MHz	20
Arduino Due	ARM Cortex-M3 (Atmel SAM3X8E), 32-bit	3.3V	512 kB	84 MHz	54
Arduino Pro Mini	ATmega328P	3.3V / 5V	32 kB	8 / 16 MHz	14
Arduino Zero	ARM Cortex-M0+ (SAMD21), 32-bit	3.3V	256 kB	48 MHz	20
Arduino Esplora	ATmega32U4	5V	32 kB 4kB bootloader	16 MHz	Integrated sensors
Arduino UNO R4 Minima	Renesas RA4M1 (Arm Cortex-M4), 32-bit	5V	256 kB	48 MHz	14
UNO R4 WiFi	Renesas RA4M1 and ESP32-S3 for Wi-Fi	5V	256 kB	48 MHz	14

1.5 Arduino Uno Host Microcontrollers

The ATmega328P of Fig.1.5 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture with [7]:

- Instruction Set: It supports 131 powerful instructions.
- Execution Speed: Most instructions execute in a single clock cycle.
- Multiplier: Includes an on-chip 2-cycle multiplier.
- Registers: Features 32 x 8 general purpose working registers.
- Flash Program Memory: 32 kBytes with a write/erase cycles of 10,000 flash.
- Timers/Counters :
    - Two 8-bit Timer/counters with separate prescaler and compare mode.
    - One 16-bit Timer/counter with separate prescaler, compare mode, and capture mode.
    - Six PWM channels.
- Analog Features :
    - 6-channel 10-bit ADC in PDIP package.
    - 8-channel 10-bit ADC in TQFP package.
    - One on-chip analog comparator.
    - Internal analog reference of 1.1V
    - Free running or single conversion mode
    - Interrupt on ADC conversion complete

- Communication Interfaces :
  - Two master/slave SPI serial interfaces.
  - One programmable serial USART.
  - 2-wire serial interface (Philips I²C compatible).

- Interrupt system :
  - 26 total interrupts (Table A.1, Annex 1).
  - 2 external pin interrupts.
- System and Security :
  - Power-on Reset (POR) and programmable Brown-out Detection (BOD).
  - Programmable watchdog timer with separate on-chip oscillator.
  - Internal calibrated oscillator.

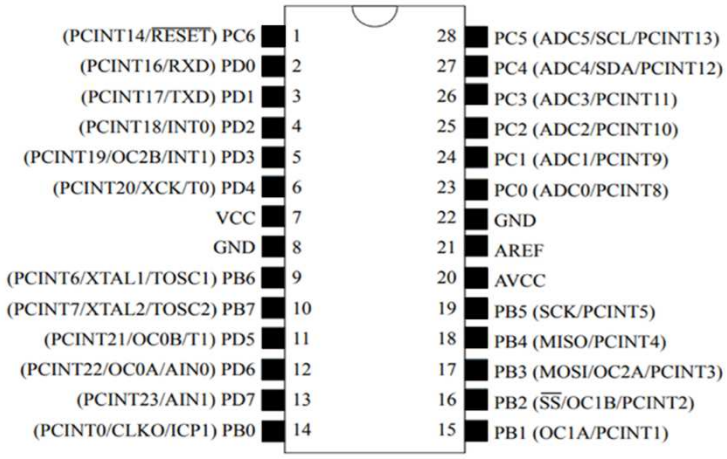


Fig.1.5: ATmega328P (PDIP) Pinout

The AVCC pin represents the supply voltage pin for the A/D Converter (ADC). It should be externally connected to VCC, even if the ADC is not used. If the ADC is used, it should be connected to VCC through a low-pass filter. Meanwhile, the analog reference voltage for the A/D Converter is connected to the AREF pin.

The ATmega328P is equipped with three, 8–bit general purpose, digital input/output (I/O) ports designated PORTB (8 bits, PORTB[7:0]), PORTC (7 bits, PORTC[6:0]), and PORTD (8 bits, PORTD[7:0]). All of these ports also have alternate functions. Therefore, each port has three registers associated with it such as:

- The Port Output Register PORTx which is used to write output data to the port.
- The Data Direction Register DDRx which is used to set a specific port pin to either output (1) or input (0).
- The Port Input Register PINx which is used to read input data from the port.

where x is the port designator (B,C, D).

The Port x data Register (PORTx):

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0x08 (0x28)	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	PORTC
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0x0B (0x2B)	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Port x Data Direction Register (DDRx):

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0x07 (0x27)	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	DDRC
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0x0A (0x2A)	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Port x Input Pins Address (PINx):

Bit	7	6	5	4	3	2	1	0	
0x09 (0x29)	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R	R	R	R	R	R	R	R	

Bit	7	6	5	4	3	2	1	0	
0x06 (0x26)	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	PINC
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

The below table represents how the DDxn and PORTxn register bits determine a General-Purpose I/O pin's mode (input/output) and whether its internal pull-up resistor is enabled, where x is the port designator (B, C, D) and n is the pin designator (0–7).

Table 1.5: General-Purpose I/O Pin Configuration

DDxn	PORTxn	I/O	Pullup
0	0	Input	No
0	1	Input	Yes
1	0	Output	No
1	1	Output	No

The second microcontroller on the Arduino Uno is the ATmega16U2, an 8-bit microcontroller that acts as a USB-to-serial interface and provides memory resources such as 16 kB ISP Flash, 512 B EEPROM, and 512 B SRAM. Most Arduino Uno clones use a dedicated chip, with FT232, CH340, CP2102 and PL2303 being the most common.

1.6 Arduino Integrated Development Environment

The C programming language provides a balance between the programmer's control of the microcontroller hardware and time efficiency in programming. The Arduino IDE is a platform for writing, compiling, and uploading code to Arduino boards [8]. It provides a user-friendly interface for

coding in the C/C++ language, specifically designed to interact with Arduino hardware. This functionality is possible because the Arduino processing boards are equipped with a bootloader program.

The Arduino IDE 2.x is open source software, and its code is hosted on GitHub. This development environment can be installed on platforms such as Windows 10 or newer (64-bit), while the legacy version (Arduino IDE 1.8.19) supports Windows 7 (32- or 64-bit) or newer (Fig.1.6). Furthermore, Arduino Cloud Editor can be accessed online from any browser for free.

The IDE of Fig.1.6 contains a text editor, a message area for displaying status, a text console, a toolbar of common functions, and an extensive menu system. The toolbar provides single-button access to the more commonly used menu features. Most of the features are self-explanatory. For instance, the "Upload" button compiles a source code (sketch) and uploads it to the Arduino board, whereas the "Serial Monitor" button opens the serial monitor feature, which allows text data to be sent to and received from the Arduino processing board (Fig.1.7). Thus, the toolbar provides a wide variety of features to compose, compile, load, and execute a sketch.

The job of the compiler is to transform the sketch provided by the user (filename.ino and maybe filename.h) into machine code (filename.hex) suitable for loading into the AVR microcontroller. Once the source files are provided to the compiler, the compiler executes two steps to render the machine code. The first step is the compilation process where source files are transformed into assembly code (filename.asm). If the program source files contain syntax errors, the compiler reports these to the user. Syntax errors are reported for incorrect use of the C programming language. An assembly language program is not generated until the syntax errors have been corrected. The assembly language source file is then passed to the assembler to transform it into machine code (filename.hex) suitable for loading to the AVR microcontroller.

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

Before uploading a sketch, it is required to select the type of Arduino board being used and the serial port it is connected to.

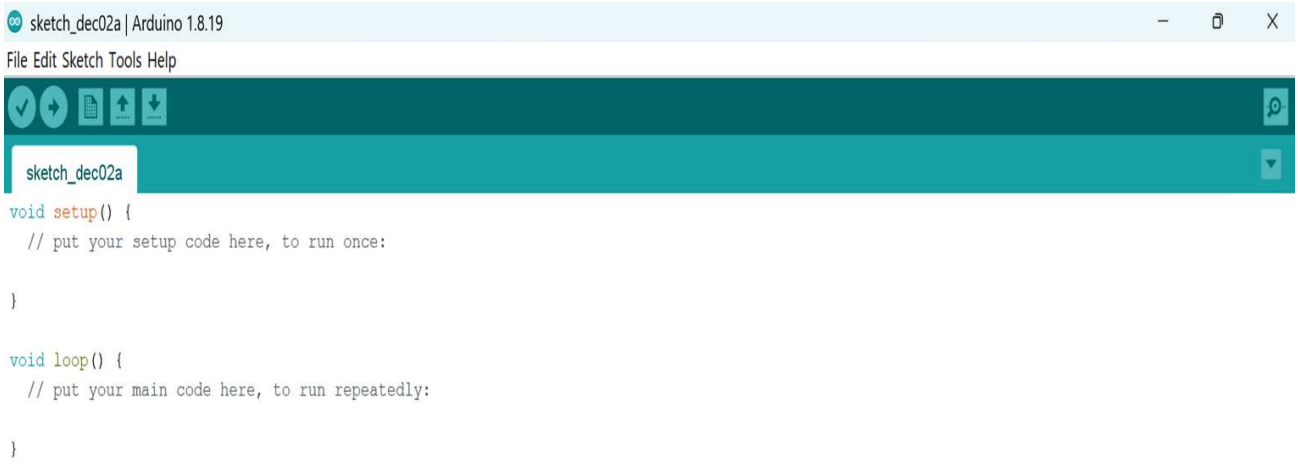


Fig.1.6: Arduino IDE Platform

Fig.1.7: Arduino IDE Toolbar Buttons

1.7 Conclusion

The goal of this chapter was to provide a tutorial on the fundamental aspects of Arduino boards (Uno and Nano) along with their associated microcontroller (ATmega328P). The functionalities and features of these components, as well as their integrated development environment, were explored to facilitate a comprehensive understanding of how to use them in various projects. The next chapter will focus on programming the Arduino Uno using the C language.

Chapter 2: Programming the Arduino Uno Board

2.1 Introduction

This chapter presents the main concepts of the Arduino-based C language, which are applied in every sketch developed by an Arduino programmer. To get the most out of Arduino, it is necessary to understand these fundamentals.

2.2 Structure of an Arduino Sketch

All Arduino sketches follow a similar structure; preprocessor directives, global variables and at least the void setup() and void loop() functions (including most of the times user defined ones). The setup() function is called once at startup to configure settings, and the loop() function continuously executes after setup finishes.

2.2.1 Preprocessor Directives

Used to include external libraries for further functionality and to define constants and macros in the sketch.

Example:

#include <library_name.h>

#define CONSTANT_NAME value

2.2.2 Program Constants and Variables

There are two types of variables used within a program: global variables and local variables. A global variable is available and accessible to all portions of the program, whereas a local variable is only known and accessible within the function where it is declared. When programming microcontrollers, it is important to know the number of bits used to store the variable and where the variable will be assigned. Therefore, the type of the variable is specified before using it, followed by its name, and an initial value if desired. Table below is a list of the data types commonly seen in Arduino (only boards based on ATmega328P and ATmega2560), with the memory size of each after the type name [6][9].

Table 2.1: Common Data types in Arduino C language

Type	Alternative Names	Size	Range
bool	boolean	1 Byte	0, 1
byte		1 Byte	0 … 255
char	signed char, int8_t	1 Byte	-128 … +127
unsigned char	uint8_t	1 Byte	0 … 255
int	signed int, int16_t	2 Byte	-32768 … +32767
unsigned int	uint16_t	2 Byte	0 … 65535
long	signed long int, int32_t	4 Byte	-2147483648 … +2147483647
unsigned long	unsigned long int, uint32_t	4 Byte	0 … 4294967295
float	double	4 Byte	-3.402823E+38 … +3.402823E+38

2.2.3 Working with Arrays

An array is a collection of data elements of the same type, chosen from the various data types available to variables. These elements are addressed by an index number that points to a specific data element. Thus, it can represent a single variable that holds multiple values.

To declare an array of a given length without initializing the values: int myArray[6];

To declare an array without explicitly choosing a size (the compiler counts the elements and creates an array of the appropriate size): int myPins[] = {2, 3, 4, 5, 6, 7};

When declaring an array of type char, it is need to make it longer by one element to hold the required the null termination character: char message[6] = "hello";
To declare an array of a given length and initialize its values: int myArray[10] = {2, 4, -8, 3, 2, 2, 7, 8, 9, 11};
An Array is zero indexed, where its first element is at index 0 (myArray[0] contains 2). For myArray with ten elements, the index nine is the last element. Hence myArray[9] contains 11 and myArray[10] is invalid and contains random information (other memory address).
To assign a value to an array: myArray[0] = 10;
To retrieve a value from an array: x = myArray[4];
Arrays are often manipulated inside for loops, where the loop counter is used as the index for each array element. For instance:

```
for (byte i = 0; i <6; i = i + 1) {  // Print the elements of myPins over the serial port
    Serial.println(myPins[i]);
  }
```

Additionally, the declaration of multi-dimensional arrays involves adding an index in square brackets for each additional dimension. Example: myArray[0][0]
To assign a value to an element in a two-dimensional array, we would just need to know the position or location of each dimension to a specific element. Example: myArray[1][2] = 42;
Furthermore, the value can be assigned to the multidimensional array at the point of its declaration. Example: int arrayTwo[2][2] = {{1, 2}, {3, 4}};

2.2.4 Pointers

A pointer points to the location in memory where the value of a variable is stored [9]. A pointer is declared by placing an asterisk (*) before its name. The pointer's type must match the type of the variable it points to.
Example:

```
int value = 10;
int *ptr;
ptr = &value;
```

where value stores the integer 10, &value gives the address of value, and ptr stores that address.
The value stored at the pointer's address can be accessed or modified using the dereference operator (*).
Example:

```
*ptr = 25;
Serial.println(*ptr);
```

Pointers are often used to pass variables by reference, allowing a function to modify the original data directly.
Example:

```
void increment(int *num) {
  *num = *num + 1;
}
int count = 5;
increment(&count);
Serial.println(count);
```

where &count passes the address of count to the function. Inside increment(), *num accesses and modifies the actual variable.

2.2.5 Program Functions

Any C program is subdivided into pieces, each with a defined action. This makes writing the program easier. There are two different pieces of code required to properly configure and call the function: the function call, and its body. Also, the function prototype provides the name of the function and any variables required by the function and any variable returned by the function, and it is provided early in the program sketch. If the function does not require variables or does not send back a variable the word "void" is placed in the variable's position.

Example:

```
int calculateAverage(int readings[], int size) {
  int sum = 0;
  for(int i = 0; i < size; i++) {
    sum += readings[i];
  }
  return sum / size;
}
```

2.2.6 Comments

Comments are used throughout the program to document what and how things were accomplished within a program. The comments will help you remember the key details of the program. Comments are not compiled into machine code for loading into the microcontroller. Comments are indicated using double slashes (//). Anything from the double slashes to the end of a line is then considered a comment. A multi-line comment can be constructed using a /* at the beginning of the comment and a */ at the end of the comment. The multi-line comment technique may be used to block out portions of code during troubleshooting.

2.3 Operators

There are many operators provided in the C language. As shown in Tables 2.2 and 2.3, the operators are grouped by general category, and the symbol, precedence, and brief description of each operator are provided [9][10]. The precedence column indicates the priority of the operator in a program statement containing multiple operators.

Within the general operations category are brackets, parenthesis, and the assignment operator. Bracket pairs are used to indicate the beginning and end of the main program or a function and to group statements in programming constructs and decision processing constructs. Furthermore, the parenthesis is used to boost the priority of an operator. The assignment operator (=) is used to assign the argument(s) on the right-hand side of an equation to the left-hand side variable.

Table 2.2: Operators precedence

Operation Type	Symbol	Precedence	Description
Arithmetic	*	3	Multiplication
	/	3	Division
	+	4	Addition
	-	4	Subtraction

Logical	<	6	Less than
	<=	6	Less than or equal to
	>	6	Greater than
	>=	6	Greater than or equal to
	==	7	Equal to
	!=	7	Not equal to
	&&	9	Logical AND
	||	10	Logical OR
General	{ }	1	Brackets, used to group program statements
	( )	1	Parenthesis, used to establish precedence
	=	12	Assignment
Conditional	?:	11	Ternary conditional (if-else shorthand)

Example:
This Arduino code reads an analog input from A0 and uses comparison, logical, and ternary operators to evaluate sensor conditions, simulate LED control, and display results on the Serial Monitor (Fig.2.1).

```
int sensorValue;        // Variable to store analog sensor reading
int threshold = 50;     // Threshold value for decision making
bool ledOn = false;     // Boolean flag to represent LED status (ON/OFF)
void setup() {
  Serial.begin(9600);   // Initialize serial communication at 9600 baud
}
void loop() {
  sensorValue = analogRead(0);   // Read analog value from pin A0 (0–1023 range)
  Serial.print("sensorValue =");
  Serial.print(sensorValue);
    Serial.print(". ");
  // --- Comparison Operator ---
    if (sensorValue >= threshold) {   // '>=' means greater than or equal to
    Serial.println("Sensor value is greater than or equal to threshold.");
  }
  if (sensorValue == 75) {          // '==' means exactly equal to
    Serial.println("Sensor value is exactly 75.");
  }
  if (sensorValue != 100) {         // '!=' means not equal to
    Serial.println("Sensor value is not 100.");
  }

  // --- Logical Operator ---
    if (sensorValue > 60 && sensorValue < 90) {  // '&&' means AND — both conditions must be true
    Serial.println("Sensor value is between 60 and 90.");
  }
  if (sensorValue < 40 || sensorValue > 70) {  // '||' means OR — at least one condition must be true
    Serial.println("Sensor value is either less than 40 OR greater than 70.");
  }
```

```
  // --- Combined Logical and Comparison ---
 if ((sensorValue >= threshold) && (sensorValue != 100)) {
    // LED turns ON only if sensorValue is >= threshold and not equal to 100
    ledOn = true;
  }
  // Print LED status using the conditional (ternary) operator
  Serial.print("LED status: ");
  Serial.println(ledOn ? "ON" : "OFF");  //condition ? value_if_true : value_if_false
  delay(1000);   // Wait 1 second before next reading
  ledOn = false; // Reset LED status for next loop iteration
}
/* Sketch uses 2344 bytes (7%) of program storage space. Maximum is 32256 bytes.
Global variables use 423 bytes (20%) of dynamic memory, leaving 1625 bytes for local variables. Maximum is 2048 bytes.*/
```

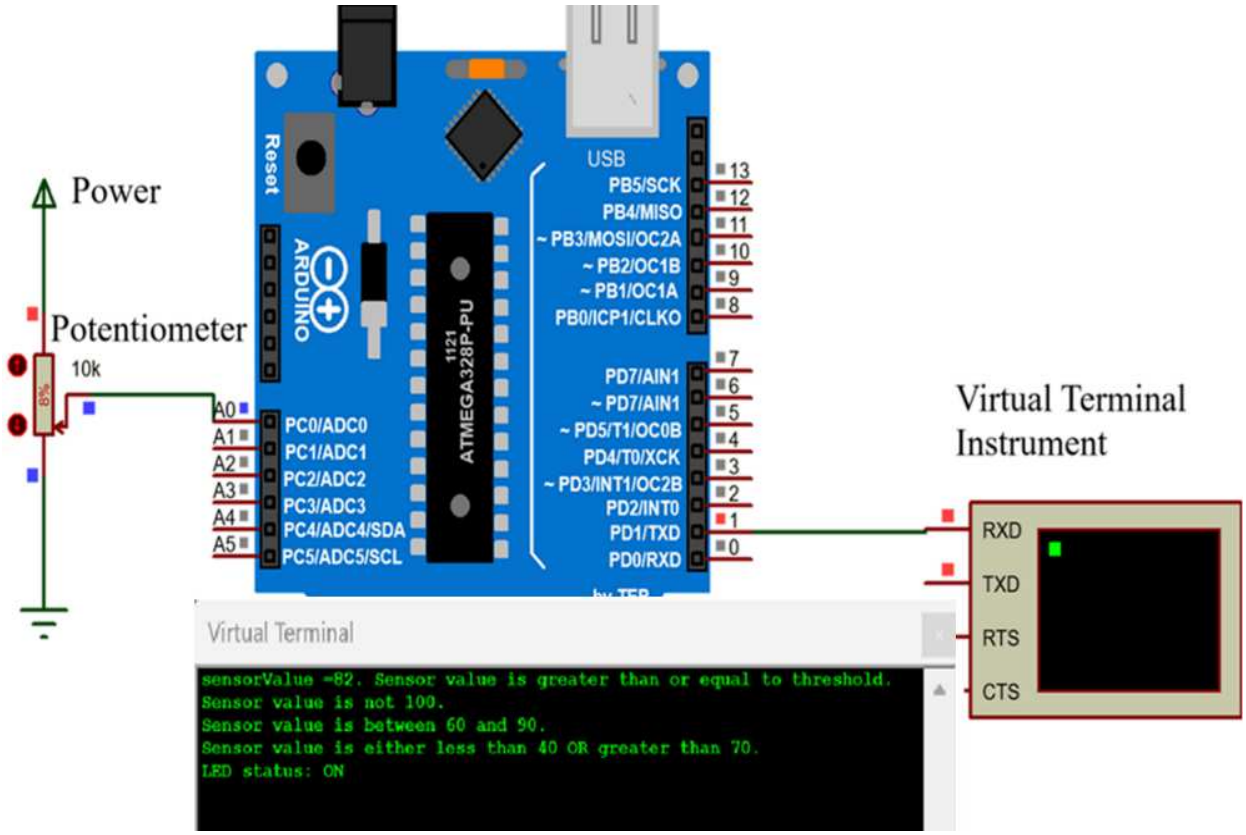


Fig 2.1: Arduino Code Validation for Comparison and Logical Operators in ISIS Proteus

There are two general types of operations in the bit manipulation category: shifting operations and bitwise operations. The details are given in Tables 2.3 and 2.4.

Example:

The following Arduino code demonstrates how bitwise and logical operators work. The simulation results are shown in Fig.2.2.

```
int a = 12;     // Binary: 00001100
int b = 5;      // Binary: 00000101
int result;     // For storing operator results
void setup() {  Serial.begin(9600);
```

```
  // --- Bitwise Shift Operators ---
 result = a << 1;   // Left shift: shift bits of 'a' left by 1 position (multiply by 2)
 Serial.print("a << 1 = ");
 Serial.println(result); // 24 (00011000)
 result = a >> 1;   // Right shift: shift bits of 'a' right by 1 position (divide by 2)
 Serial.print("a >> 1 = ");
 Serial.println(result); // 6 (00000110)
 // --- Bitwise AND ---
 result = a & b;    // Bitwise AND: only 1 where both bits are 1
 Serial.print("a & b = ");
 Serial.println(result); // 4 (00000100)
 // --- Bitwise OR ---
 result = a | b;    // Bitwise OR: 1 where either bit is 1
 Serial.print("a | b = ");
 Serial.println(result); // 13 (00001101)
 // --- Bitwise XOR ---
 result = a ^ b;    // Bitwise XOR: 1 where bits are different
 Serial.print("a ^ b = ");
 Serial.println(result); // 9 (00001001)
// --- Bitwise NOT ---
 result = ~a;       // Bitwise NOT: inverts all bits
 Serial.print("~a = ");
 Serial.println(result); // -13 (in 2's complement form)
 // --- Logical NOT ---
 bool flag = true;
 Serial.print("flag = ");
 Serial.println(flag);
 Serial.print("!flag = ");
 Serial.println(!flag); // Logical NOT flips true/false
 Serial.println("----------------------");
}
void loop() {//do nothing
 }
/* Sketch uses 2128 bytes (6%) of program storage space. Maximum is 32256 bytes.
Global variables use 276 bytes (13%) of dynamic memory, leaving 1772 bytes for local variables. Maximum is 2048 bytes.*/
```

Table 2.3: Bit manipulation and unary Operators

Operation Type	Symbol	Precedence	Description
Bit Manipulation	<<	5	Shift left
	>>	5	Shift right
	&	8	Bitwise AND
	^	8	Bitwise exclusive OR
	|	8	Bitwise OR
Unary	!	2	Unary negative

	~	2	One's complement (bit-by-bit inversion)
	++	2	Increment
	--	2	Decrement
	type(argument)	2	Casting operator (data type conversion) Example: (int)a converts a to an integer

Table 2.4: Bitwise operators

Syntax	Description	Example
a | b	Bitwise OR	PORTB |=0x80 ; //turn on bit 7 (msb)
a & b	Bitwise AND	if ((PINB & 0x81) == 0) // check bit 7 and bit 0
a ^ b	Bitwise Exclusive OR	PORTB ^= 0x80; // toggle/flip bit 7
~a	Bitwise Complement	PORTB &= ~0x80; // turn off bit 7

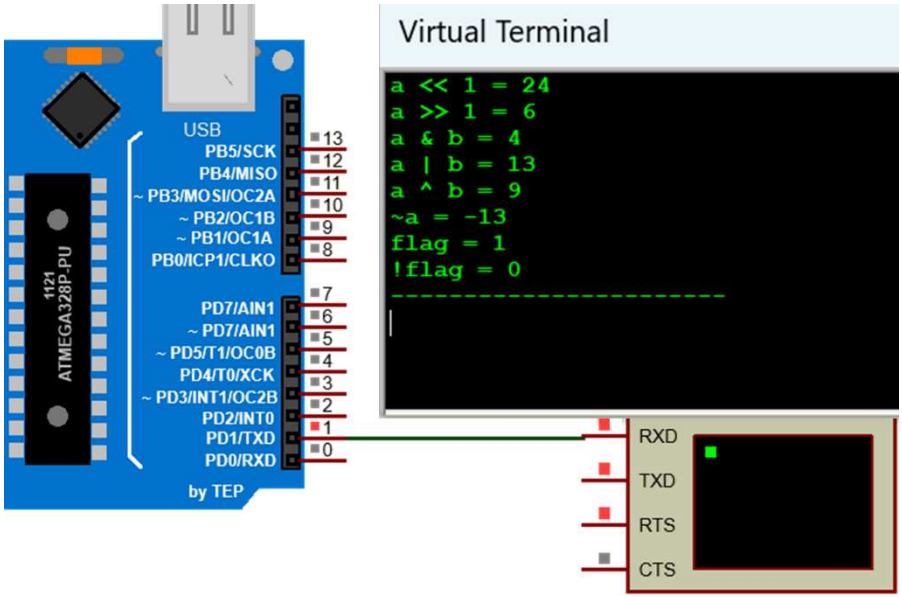


Fig.2.2: Validation of Arduino Bitwise and Logical Operations in ISIS Proteus

2.4 Looping Statements in Arduino

2.4.1 For Loop

Repeats a block of code a set number of times [9][11].

Example:

```
void setup() {
  Serial.begin(9600); // start serial communication at 9600 bps
}
void loop() {
  for(int i = 1; i <= 5; i++) {
    Serial.println(i);  // prints 1 2 3 4 5
    delay(500);         // half-second delay
  }
    while(1); // stop after printing once
}
```

2.4.2 While Loop

Executes a block of code as long as the condition is true.

Example:

```
void setup() {
  Serial.begin(9600);
}
void loop() {
  int i = 1;
  while(i <= 5) {
    Serial.println(i);
    i++;
    delay(500);
  }
}
```

2.4.3 Do-While Loop

Executes at least once, then checks the condition.

Example:

```
void setup() {
  Serial.begin(9600);
}
void loop() {
  int i = 1;
  do {
    Serial.println(i);
    i++;
    delay(500);
  } while(i <= 5);
}
```

2.5 Decision-Making in Arduino

2.5.1 If-Else

An if–else statement executes the if block when the condition is true, and executes the else block when the condition is false.

Example:

```
void setup() {  Serial.begin(9600);
}
void loop() {
  int sensorValue = analogRead(A0); // read analog pin A0
  if(sensorValue > 512) {     //if block of code
    Serial.println("Value is high");
  } else { //else block of code
    Serial.println("Value is low");
  }
```

```
delay(1000);}
```

2.5.2 Switch-Case

Allows multiple conditional branches based on a variable's value.
Example:

```
void setup() {
  Serial.begin(9600);
}
void loop() {
  boolean buttonState = digitalRead(2); // read digital pin D2
  switch(buttonState) {
    case HIGH:
      Serial.println("Button pressed");
      break;
    case LOW:
      Serial.println("Button not pressed");
      break;  }
  delay(500);
}
```

2.6 Structure

A structure (struct) allows grouping variables of different types under a single name [9]. By using the typedef keyword, it is unnecessary to write struct each time a variable of that type is declared.
Example:

```
typedef struct { // Define a custom structure named 'SensorData' to group related sensor values
  float temperature;          // Variable to store temperature in °C
  float humidity;             // Variable to store humidity percentage
  unsigned long timestamp;    // Variable to store the time (in milliseconds) when the data was recorded
} SensorData;
SensorData reading; // Create a variable 'reading' of type SensorData
void setup() {
  Serial.begin(9600); // Start serial communication at 9600 baud rate
  // Assign sample values to the 'reading' structure
  reading.temperature = 23.5;       // Set temperature value
  reading.humidity = 60.2;          // Set humidity value
  reading.timestamp = millis();     // Get the current time since program started (in milliseconds)
  // Print the values stored in the structure to the Serial Monitor
  Serial.print("Temp: ");
  Serial.println(reading.temperature);
  Serial.print("Humidity: ");
  Serial.println(reading.humidity);
  Serial.print("Time: ");
  Serial.println(reading.timestamp);
}
```

```
void loop() {  // To continuously update the data structure with incoming sensor readings.
}
```

2.7 Programming the ATmega328P Bootloader via Arduino

Fig. 2.3 shows the standard wiring setup for burning the bootloader to an ATmega328P using an Arduino as an In-System Programmer [12][13]. This process is explained through the following steps:

1. Upload ArduinoISP sketch
   - In the Arduino IDE:
     - Go to File > Examples > 11. ArduinoISP > ArduinoISP
     - Select the appropriate Arduino board from Tools > Board
     - Select the correct COM Port under Tools > Port
     - Click Upload
2. It is necessary to follow the table and use the required components below to complete the circuit connections (Table 2.5).
   - An external 16 MHz crystal oscillator with two 22pF capacitors connected to GND (if not using internal clock).
   - A 10kΩ pull-up resistor between RESET (Pin 1) and Vcc.
   - The AREF (Pin 21) is connected to Vcc for analog stability (optional).

Table 2.5: Arduino to ATmega328P ISP/SPI Pin Mapping

Arduino Pin	ATmega328P Pin	Function
D10	Pin 1 (RESET)	Reset
D11	Pin 17 (PB3/MOSI)	SPI MOSI
D12	Pin 18 (PB4/MISO)	SPI MISO
D13	Pin 19 (PB5/SCK)	SPI Clock
5V	Pin 7 & 20	Vcc & AVcc
GND	Pin 8 & 22	Ground

3. Set the board type
   - In Tools > Board, choose:
     - Arduino Duemilanove or Nano with ATmega328 (if using an external 16 MHz crystal).
4. Set the programmer
   - Go to Tools > Programmer > Arduino as ISP
5. Burn the bootloader
   - Go to Tools > Burn Bootloader
   - Wait until the following message is displayed: “Done burning bootloader” in the status bar.
   - Now the jumper wires can be removed.

Sketches can be uploaded by either placing the chip into an Arduino board or using a USB-to-Serial adapter.

After the bootloader is burned, the ATmega328P is typically removed from the breadboard and placed into an Arduino Uno. Then, the ATmega328P can be programmed via serial communication (over USB). Instead of using an external 16 MHz crystal and 18-22 picofarad capacitors, it is possible to configure the ATmega328P to use its internal 8 MHz RC oscillator as a clock source instead.

2.8 Using a Serial Programmer via ICSP

The below diagram shows the essential connections for using a serial programmer (USBasp with AVRDUDE software, or the Arduino used as an ISP) to communicate with and program the ATmega328P via its In-Circuit Serial Programming interface. It connects the microcontroller to an external programmer, showing connections for MISO, SCK, MOSI, RESET, and VCC/GND. Also, this diagram is for directly programming the ATmega328P via its ICSP pins, while the previous diagram (Fig.2.4) involves using an Arduino to burn the bootloader by connecting through the serial interface [10].

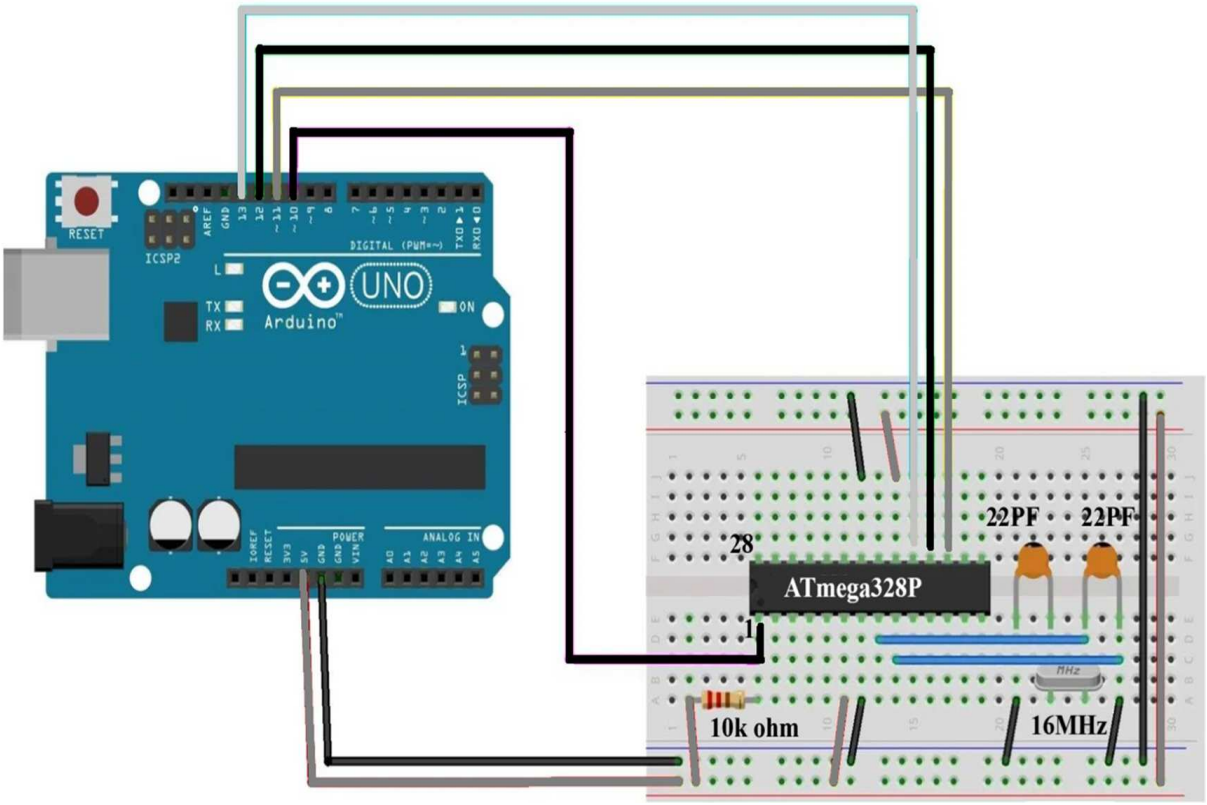


Fig2.3: Burning Bootloader to ATmega328P on Breadboard

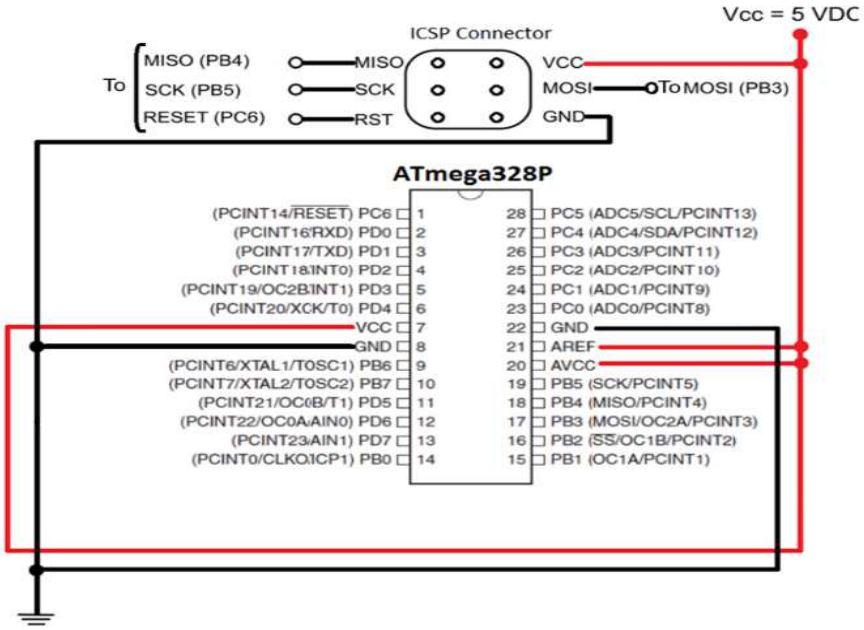


Fig.2.4: In Serial Programmer Connections to ATmeta328P

2.9 Conclusion

The current chapter has covered in detail the basics of Arduino programming in terms of the structure of an Arduino sketch, the concept of variables, functions, operators, loops, and decision-making statements that serve to implement logical control actions in programming. Therefore, the subject matter of the next chapter shall introduce readers to the basic analog and digital I/O operations on the Arduino to perform real-world tasks with the help of external devices such as sensors, switches, and actuators connected to the Arduino board.

Chapter 3: Arduino Digital and Analog I/O Operations

3.1 Introduction

The Arduino Uno features both digital and analog input/output (I/O) capabilities, allowing it to interact with various electronic shields and sensors. Thus, this chapter discusses practical applications and functions of digital and analog I/O operations.

3.2 Pin Mode Configuration with pinMode()

This function is employed to set the given pin as input/output and requires two arguments: the pin and the mode. The first argument helps to configure the pin we want to set as input/output. While the other can be set as input (INPUT) or output (OUTPUT) or input pull-up (INPUT_PULLUP) [6][9][14].

INPUT:

This parameter set any digital pin as a high-impedance input to get the logic-level voltage from sensors or buttons. An external resistor can be used as a pull-down or a pull-up.

OUTPUT:

It sets any digital pin as a low impedance output to drive LEDs, relays, and other actuators. Its value can be HIGH or LOW using the digitalWrite() function.

INPUT_PULL:

It can be used to connect basic buttons or switches without the use of additional pull-up resistors (internal pull-up resistor (~20k-50kΩ)). This means the pin level can be considered HIGH if not grounded and LOW if grounded.

3.3 Digital and Analog I/O Functions

The Arduino C language has numerous functions used when dealing with digital and analog pins. These functions enable the ability to set and get values of pins, output the analog signal, among other functions. Below are the main functions used in the basic language [6][9][14].

Table 3.1: Digital and Analog Pin Functions and Their Usage

Function	Usage	Pin Type	Value Range	Description
digitalWrite(Pin,Value)	Set digital pin output	Digital pins	HIGH / LOW	Set pin HIGH or LOW voltage
digitalRead(Pin)	Read digital pin state	Digital pins	HIGH / LOW	Read pin state
analogRead(Pin)	Read analog voltage	Analog input pins	0 - 1023	Convert analog voltage to digital
analogWrite(Pin, Duty cycle)	Simulate analog output (PWM)	PWM digital pins	0 - 255	Produces a PWM signal for simulating analog output

Example:

This code reads the state of a button with an external pull-up resistor and controls an LED based on the button press (Fig 3.1). It also reads an analog sensor value (voltage across the potentiometer) and uses it to adjust the brightness of a PWM-controlled LED.

```
// Pin Definitions
const int buttonPin = 2;   // Pin where the button is connected
const int ledPin = 12;     // Pin where the LED is connected
const int analogPin = A0;  // Pin where the potentiometer is connected
const int pwmPin = 3;      // Pin where PWM is used to control LED brightness

void setup() {
  // Initialize the buttonPin as an input with and without pull-up
  pinMode(buttonPin, INPUT);          // Without pull-up resistor
```

```
 //pinMode(buttonPin, INPUT_PULLUP);   // With pull-up resistor

  // Initialize the LED pin as an output
  pinMode(ledPin, OUTPUT);

  // Initialize PWM pin for analogWrite
  pinMode(pwmPin, OUTPUT);

  // Start the serial communication for debugging
  Serial.begin(9600);
}

void loop() {
  // Reading digital input with pull-up enabled
  int buttonState = digitalRead(buttonPin);  // Read the button state

  // If the button is pressed (assuming pull-up configuration)
  if (buttonState == LOW) { // LOW because the input is pulled high when not pressed
    digitalWrite(ledPin, HIGH);  // Turn on the LED
  } else {
    digitalWrite(ledPin, LOW);   // Turn off the LED
  }

  // Reading analog input from a sensor (e.g., a potentiometer)
  int sensorValue = analogRead(analogPin);   // Read the analog value from the sensor
  Serial.print("Analog value: ");
  Serial.println(sensorValue);  // Print the sensor value (0 to 1023)

  // Use the analogRead value to adjust the PWM signal on the PWM pin
  // Map the sensor value (0-1023) to PWM range (0-255)
  int pwmValue = map(sensorValue, 0, 1023, 0, 255);
  analogWrite(pwmPin, pwmValue);   // Set the PWM output (LED brightness)

  // Add a small delay to avoid excessive serial printing
  delay(100);
}
/* Sketch uses 2672 bytes (8%) of program storage space. Maximum is 32256 bytes.
Global variables use 202 bytes (9%) of dynamic memory, leaving 1846 bytes for local variables. Maximum is 2048 bytes.*/
```

3.4 LED Blinking Examples Using Arduino

Four simple examples are discussed to illustrate the different approaches to blink the built-in LED. The first two examples use the delay function to switch the LED ON and then OFF, causing the code execution to wait during the delay period. The third example replaces delay with the millis function, which lets the Arduino continue doing other tasks simultaneously while the LED is blinking. The fourth code uses timer and an interrupt to toggle the LED ON and OFF at predefined time intervals. Each code switches the LED connected to pin PB5 (Pin D13) ON and OFF every one second.

3.4.1 LED Blink Using delay() Function (Direct Register Manipulation)

In this example, the LED connected to pin PB5 blinks every second using direct register manipulation. The delay() function pauses the program for the specified time (in milliseconds).

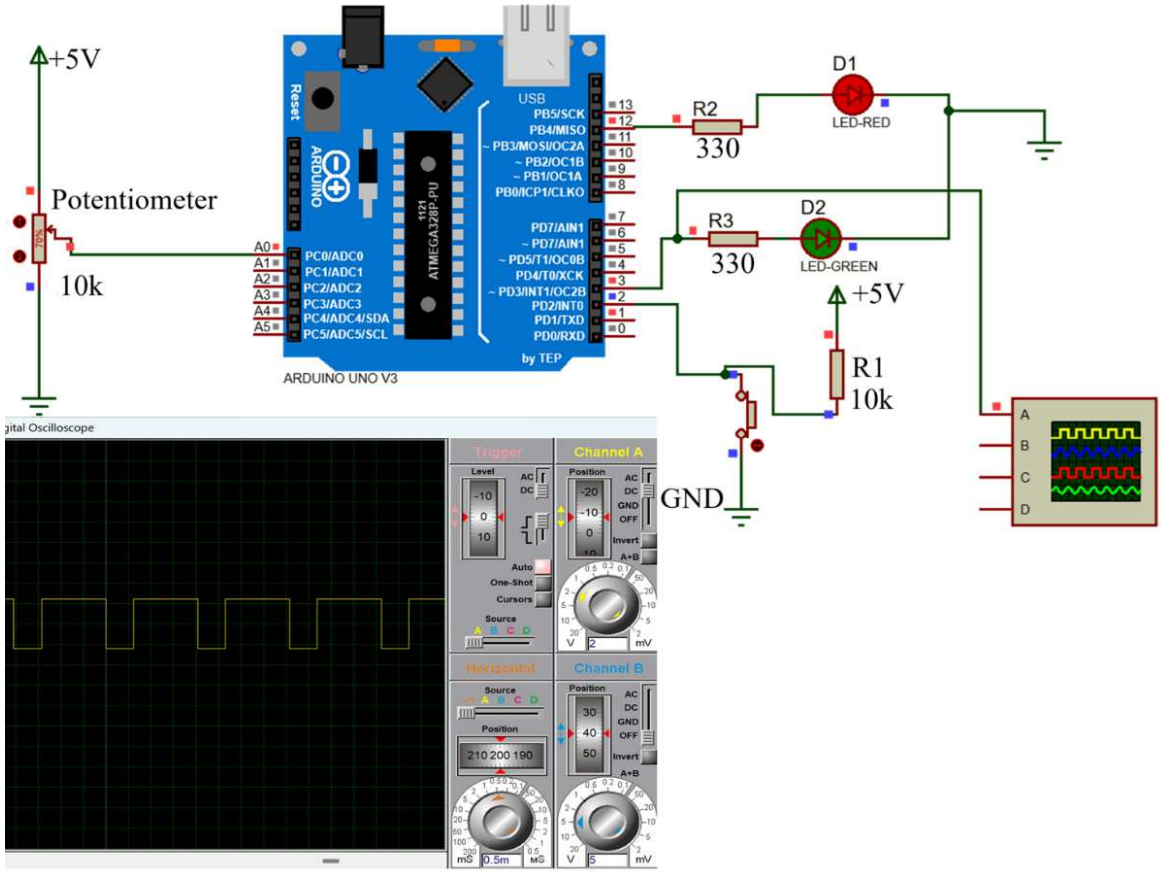


Fig.3.1: Digital and Analog Input/Output Test

```
const int led_pin = PB5;
void setup()
{// Set pin 13 (PB5) as output
  DDRB |= (1 << PB5);
}
void loop() {
    // Turn LED on
    PORTB |= (1 << PB5);
    delay(1000); // Wait for 1 second

    // Turn LED off
    PORTB &= ~(1 << PB5);
    delay(1000); // Wait for 1 second
}
/*Sketch uses 640 bytes (1%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables.
Maximum is 2048 bytes.   */
```

3.4.2 LED Toggle Using XOR Operator

This version toggles the LED ON and OFF every second using the XOR (^) operator for direct register manipulation.

```
const int ledpin = PB5;
void setup() {
```

```
  // put your setup code here, to run once:
DDRB |=(1<<ledpin);
}

void loop() {
  // put your main code here, to run repeatedly:
PORTB ^=(1<<ledpin);
delay(1000);
}
/*Sketch uses 602 bytes (1%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is
2048 bytes. */
```

3.4.3 Non-Blocking LED Blink Using millis() Function

This method allows the LED to blink while the Arduino can perform other actions. The millis() function returns the number of milliseconds since the board started running its current sketch [8].

```
int previousLEDstate = 0;        // 0 = LOW, 1 = HIGH
unsigned long lastTime = 0;      // Stores last time LED toggled
int interval = 1000;             // Blink interval (milliseconds)

void setup() {
  DDRB |= (1 << PB5);            // Set PB5 (Arduino pin 13) as output
}
void loop() {
  unsigned long currentTime = millis();  // Get current time

  if (currentTime - lastTime >= interval) {
    lastTime = currentTime;              // Update last toggle time

    // Toggle LED state
    if (previousLEDstate) {
      PORTB &= ~(1 << PB5);              // Turn LED OFF (clear bit)
      previousLEDstate = 0;
    } else {
      PORTB |= (1 << PB5);               // Turn LED ON (set bit)
      previousLEDstate = 1;
    }
  }
}
/* Sketch uses 564 bytes (1%) of program storage space. Maximum is 32256 bytes.
Global variables use 15 bytes (0%) of dynamic memory, leaving 2033 bytes for local variables.
Maximum is 2048 bytes.*/
```

3.4.4 LED Blink Using Timer1 Overflow Interrupt

This example toggles the LED every second using Timer1 overflow interrupts with a prescaler of 256, achieving non-blocking blinking without using delay() [15].

```
const int led_pin = PB5;
void setup() {
DDRB |=(1<<PORTB5);
TCCR1A=0;
TCCR1B=0;
TCNT1=3036;
```

```
TCCR1B|=(1<<CS12);//prescaler 256
TIMSK1|=(1<<TOIE1);//enable timer overflow
sei();
}

ISR(TIMER1_OVF_vect)
{ PORTB ^=(1<<led_pin);
  TCNT1=3036;
}
void loop() {//do nothing
}
/* Sketch uses 536 bytes (1%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables.
Maximum is 2048 bytes.*/
```

3.5 Analog to Digital Conversion Module

This section describes the main ADC control and data registers. It also explains how the analog reference voltage is selected and discusses how the conversion time depends on the ADC clock and prescaler settings [7][10].

The following registers control and store ADC settings and results.

3.5.1 ADC Registers

ADMUX: It represents the ADC multiplexer selection register. It is used to select input channel, reference voltage, and result alignment.

Bit	7	6	5	4	3	2	1	0
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Bits:

- REFS [1:0]: Reference selection (00=AREF, 01=AVCC, 11=Internal 1.1V, 10=Reseved).
- ADLAR : Result alignment (0=Right, 1=Left).
- MUX [3:0]: Input channel selection (0000 selects channel 0, 0001 selects channel 1, and so on).

ADCSRA: It stands for the ADC Control and Status Register A. It Controls enabling, start, auto-trigger, interrupt, and prescaler.

Bit	7	6	5	4	3	2	1	0
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Bits:

- ADEN : Enable ADC.
- ADSC : Start conversion.
- ADATE : Auto Trigger Enable.
  - 0 : conversions only start when ADSC is set manually.
  - 1 : conversions are triggered automatically based on ADTS [2:0].
- ADIF : ADC Interrupt Flag
  - Set when conversion completes.
  - Cleared automatically when ISR runs, or manually by writing a 1.
- ADIE : ADC Interrupt Enable

0 : no interrupt request.
1 : ADC completion triggers interrupt when ADIF is set.

- ADPS [2:0] : Prescaler select (division factor 2–128)

Common prescaler values are shown in the table below.

Table 3.2: ADC Clock Prescaler Values

ADPS2:0	Division Factor
000	2
001	2
010	4
011	8
100	16
101	32
110	64
111	128

ADCL and ADCH: ADC Data Registers.

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
(0x79)	–	–	–	–	–	–	ADC9	ADC8	ADCH
(0x78)	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
(0x79)	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
(0x78)	ADC1	ADC0	–	–	–	–	–	–	ADCL

ADCSRB: Control and Status Register B.

Bit	7	6	5	4	3	2	1	0
(0x7B)	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0
Read/Write	R	R/W	R	R	R	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Bits:
- ACME : Analog Comparator Multiplexer Enable.
- ADTS [2:0] : When ADATE=1, it allows auto trigger source select (000=free running mode, 001=analog comparator, 010=external interrupt, 011=Timer/Counter0 compare match A, etc.)

DIDR0: Digital Input Disable Register 0.

Bit	7	6	5	4	3	2	1	0
(0x7E)	–	–	ADC5D	ADC4D	ADC3D	ADC2D	ADC1D	ADC0D
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Bits:
ADC5D–ADC0D: Disable digital input buffer on ADC pins (saves power).

Note
A single conversion is started by writing a '0' to the Power Reduction ADC bit in the Power Reduction Register (PRR.PRADC initially equal to 0), and writing a '1' to the ADC Start Conversion bit in the ADC Control and Status Register A (ADCSRA.ADSC). ADCS will stay high as long as the conversion is in progress, and will be cleared by hardware when the conversion is completed. If a different data channel is selected while a conversion is in progress, the ADC will finish the current conversion before performing the channel change. Alternatively, a conversion can be triggered automatically by various sources. Auto triggering is enabled by setting the ADC Auto Trigger Enable bit (ADCSRA.ADATE). The trigger source is selected by setting the ADC Trigger Select bits in the ADC Control and Status Register B (ADCSRB.ADTS).

3.5.2 Analog Reference Function
This function defines the maximum voltage that can be sensed and configures the reference voltage used for analog inputs. For Arduino boards based on the ATmega328P, the analogReference() function accepts a single parameter, which can be one of the following keywords:

- DEFAULT: 5V on 5V Arduino boards or 3.3V on 3.3V Arduino boards.
- INTERNAL**:** a built-in reference, equal to 1.1V on the ATmega328P and 2.56V on the ATmega32U4.
- INTERNAL1V1: a built-in 1.1 VDC reference (Arduino Mega only).
- EXTERNAL: the voltage applied to the AREF pin (0V to 5V only) is used as the reference.

3.5.3 Conversion Rate
The conversion time is the duration required for sampling an analog input voltage and producing a corresponding digital output value. For the ATmega328P microcontroller:

- The ADC uses a successive approximation method.
- It requires 13 ADC clock cycles to complete one conversion.

Thus, it is calculated by the following equation:

$$\text{ADC Conversion Time} = \left(\frac{13}{f_{ADC}}\right) \qquad (3.1)$$

where:

- f_{ADC} is the ADC clock frequency (must be between 50 kHz and 200 kHz for full 10-bit resolution) [7].
- The ADC clock is derived from the system clock (typically 16 MHz on an Arduino Uno) divided by a prescaler.

If the ADC clock is set to 125 kHz (via a prescaler of 128) then the conversion time will be 104 µs (13cycles/125,000kHz). By default, the successive approximation circuitry requires an input clock frequency between 50kHz and 200kHz to get maximum resolution. If a lower resolution than 10 bits is needed, the input clock frequency to the ADC can be higher than 200kHz to get a higher sample rate [7].
Example:
This Arduino code measures and reports the average ADC conversion time (in microseconds) for 100 consecutive analog readings from pin A0, using direct register manipulation with a 125 kHz ADC clock (prescaler 128) and AVcc reference (5V), printing results to serial every second. Also, commented alternative shows 52µs expected time with 250 kHz clock (prescaler 64).

```
#define NUM_SAMPLES 100          // Number of samples to take
void setup() {
  // Initialize Serial communication for output at 57600 baud rate
Serial.begin(57600);
```

```
  // ADC Setup: Enable ADC, set prescaler to 128 (Fadc=16 MHz / 128 = 125 kHz ADC clock)
  //Conversion time = 13/125000=104uS.  ATmega ADC takes 13 ADC clock cycles for single
//conversion. After uploading the sketch code, we get 108uS.
  ADCSRA = (1 << ADEN)  // ADC Enable
       | (1 << ADPS2) | (1 << ADPS1)|(1<<ADPS0); // Prescaler of 128

  // ADC Setup: Enable ADC, set prescaler to 64 (Fadc=16 MHz / 64 = 250 kHz ADC clock)
  //Conversion time = 13/250000=52uS. After uploading the sketch code, we get 56uS.
  //ADCSRA = (1 << ADEN)  // ADC Enable
  //       | (1 << ADPS2) | (1 << ADPS1); // Prescaler of 64
  // Set the reference voltage to AVcc (5V) and select ADC channel 0 (A0 pin)
  ADMUX = (1 << REFS0); // REFS0 set to 1 for AVcc, MUX bits default to 0000 for channel 0
}
void loop() {
unsigned long startTime, endTime;
unsigned long totalConversionTime = 0;
  // Take 100 samples and measure the total conversion time
for (uint8_t i = 0; i < NUM_SAMPLES; i++) {
    // Start ADC conversion by setting ADSC bit in ADCSRA
    ADCSRA |= (1 << ADSC);   // ADSC = 1, start conversion
    // Start timing
startTime = micros();
    // Wait until the conversion completes (ADSC becomes 0 when done)
while (ADCSRA & (1 << ADSC));
    // End timing
endTime = micros();
    // Add the conversion time for this sample
totalConversionTime += (endTime - startTime);
  }

  // Output results to the serial monitor
Serial.print("Average Conversion Time: ");
Serial.print(totalConversionTime / NUM_SAMPLES);
Serial.println(" us");

  // Delay for readability
delay(1000);
}
/* Sketch uses 1912 bytes (5%) of program storage space. Maximum is 32256 bytes.
Global variables use 218 bytes (10%) of dynamic memory, leaving 1830 bytes for local variables.
Maximum is 2048 bytes.*/
```

3.6 LCD Display Interface

The LiquidCrystal library allows to control LCD displays that are compatible with the Hitachi HD44780 driver (14 or 16-pin interface). The interface consists of the following pins [8][16][17]:

- A Register Select (RS) pin determines where in the LCD's memory data is being written. It can select either the data register, which holds what goes on the screen, or an instruction register, which is where the LCD's controller looks for instructions on what to do next.
- A Read/Write (R/W) pin that selects reading mode or writing mode
- An Enable pin that enables writing to the registers

- 8 data pins (D0 -D7)**.** The states of these pins (high or low) represent the bits being written to or read from a register.

There are also a display contrast pin (Vo), power supply pins (+5V and GND), and LED backlight pins (Bklt+ and Bklt-) used to power the LCD, control the display contrast, and turn the LED backlight ON and OFF, respectively. The process of controlling the display involves placing the data that forms the image to be displayed into the data registers, followed by loading instructions into the instruction register. The LiquidCrystal library simplifies this process, eliminating the need to understand the low-level instructions.

Hitachi-compatible LCDs can be controlled in two modes: 4-bit or 8-bit. The 4-bit mode requires seven I/O pins from the Arduino, while the 8-bit mode requires eleven pins (Fig. 3.2). For displaying text on the screen, most operations can be performed in 4-bit mode. Thus, example N°13 shows how to control a 16x2 LCD using 4-bit mode.

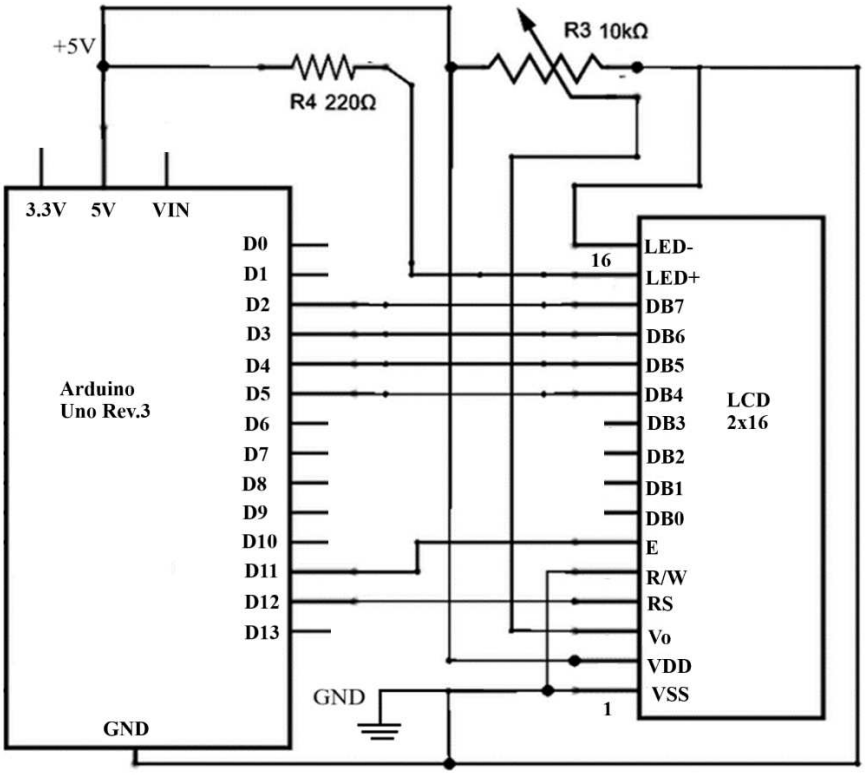


Fig.3.2: 16x2 LCD Display Controlled by Arduino Uno in 4-bit Mode

Example:

This Arduino code reads two analog voltages (A0 and A1), averages 20 samples per channel, calculates voltages, displays them on a 16x2 LCD (4-bit mode) as "V1: X.XXV" and "V2: X.XXV", and prints them to serial (e.g., "Voltage 1: X.XXV Voltage 2: X.XXV") every 100ms while clearing the LCD between updates (Fig.3.3).

```
#include <LiquidCrystal.h>   // include the library code:
// initialize the library by associating any needed LCD interface pin
// with the arduino pin number it is connected to
const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
void setup()
  {
 // set up the LCD's number of columns and rows:
```

```
lcd.begin(16, 2);
Serial.begin(57600);
}
void loop()
{ float val1=0.00, val2=0.00;
  int data1=0, data2=0;
for(byte i=1;i<=20;i++){
data1 += analogRead(A0);
data2 += analogRead(A1);
}
val1= data1/4096.0;
val2= data2/4096.0;
Serial.print("Voltage 1: ");
Serial.print(val1);
Serial.print("V");
Serial.print("   Voltage 2: ");
Serial.print(val2);
Serial.println("V");
lcd.setCursor(0, 0);
lcd.print("V1: ");
lcd.print(val1);
lcd.print("V");
lcd.setCursor(0, 1);
lcd.print("V2: ");
lcd.print(val2);
lcd.print("V");
delay(100);
lcd.clear();
}
/* Sketch uses 4916 bytes (15%) of program storage space. Maximum is 32256 bytes.
Global variables use 274 bytes (13%) of dynamic memory, leaving 1774 bytes for local variables.
Maximum is 2048 bytes.*/
```

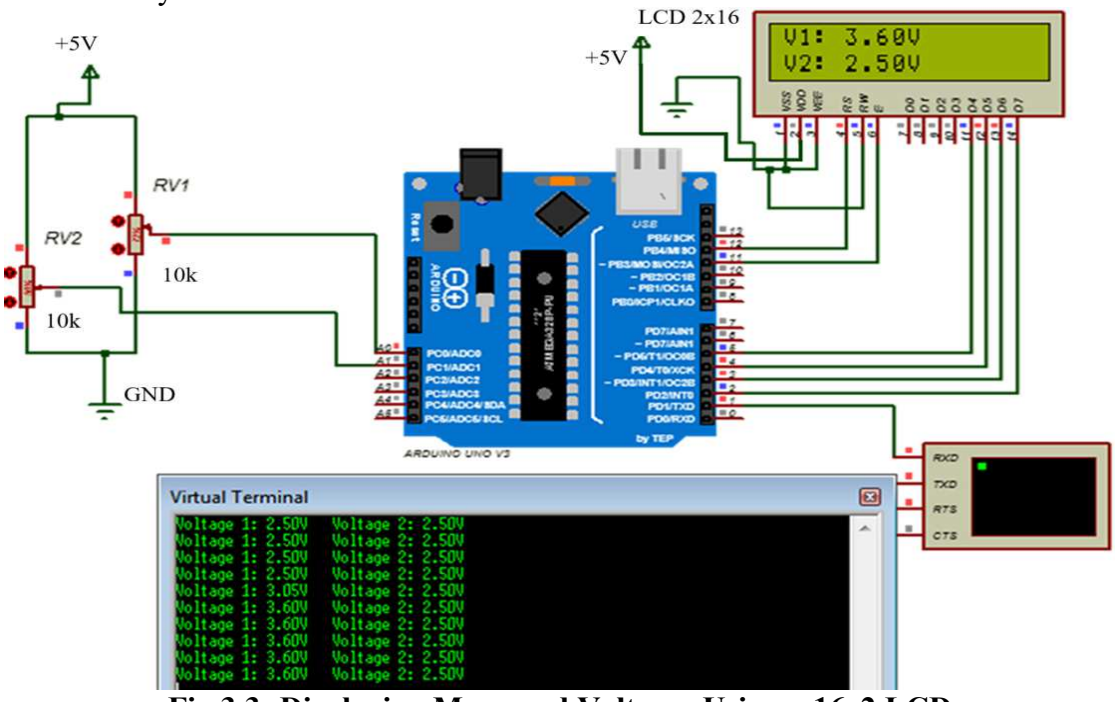


Fig.3.3: Displaying Measured Voltages Using a 16x2 LCD

3.7 Temperature Measurement Using LM35 Sensor

In Fig.3.4, the Arduino code reads temperature data from an LM35 sensor (output of 10 mV/°C) using direct ADC register manipulation with the 1.1V internal reference [6][8]. It averages 100 samples and displays the result both on the Serial Monitor (as 'Temperature: XX.XX°C') and on a 16x2 LCD (as 'Temp: XX.XX°C'), including proper degree symbols, with updates every 100ms. Accordingly, Fig.3.5 shows the practical setup, which includes a 10 kΩ potentiometer to adjust the LCD contrast.

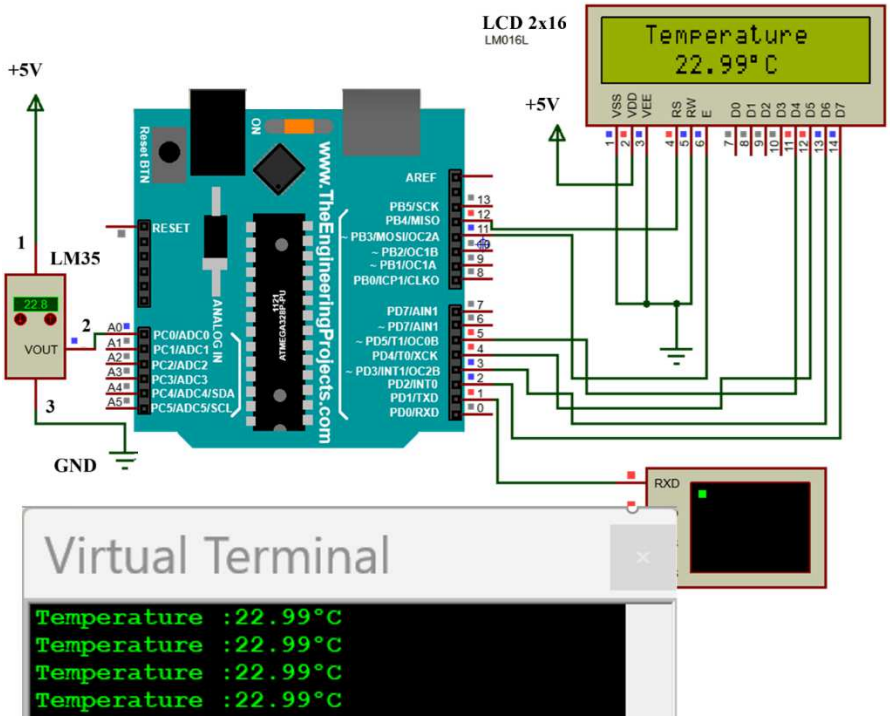


Fig.3.4: Temperature monitoring

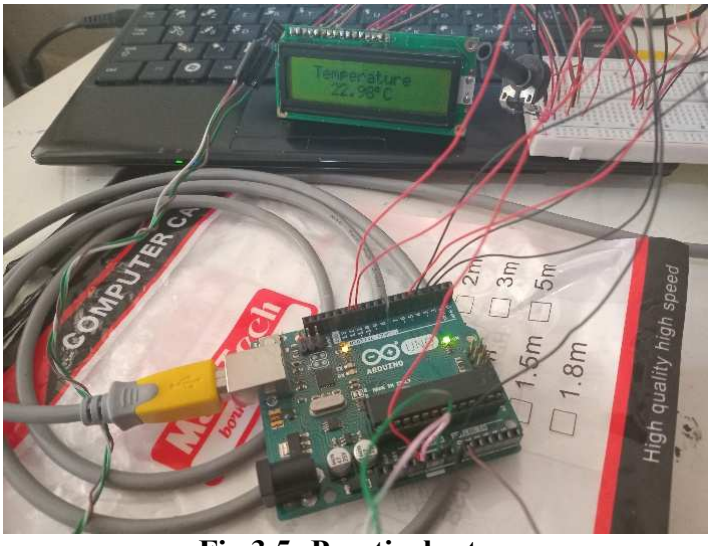


Fig.3.5: Practical setup

```
#include <LiquidCrystal.h>
// initialize the library by associating any needed LCD interface pin
// with the arduino pin number it is connected to
const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
```

```
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
const float VREF = 1.10;   // volts at AREF pin (CHANGE THIS!)
const byte ADC_PIN = 0;     // ADC0 = A0
const uint8_t SAMPLES = 100;
void setup() {
  Serial.begin(57600);
  lcd.begin(16, 2);
  // --- ADC setup ---
  ADMUX =
    (1 << REFS1) | (1 << REFS0) |  // Internal 1.1 V reference
    (ADC_PIN & 0x0F);              // ADC channel
  ADCSRA =
    (1 << ADEN)  |                 // Enable ADC
    (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0); // Prescaler 128
  DIDR0 |= (1 << ADC0D);           // Disable digital input on ADC0
  // discard first result
  startADC();
  waitADC();
  (void)ADC;
}
void loop() {
  uint32_t sum = 0;
  for (uint8_t i = 0; i < SAMPLES; i++) {     sum += readADC();  }
  float avg = sum / (float)SAMPLES;
  // Convert ADC → voltage → temperature
  float voltage_mV = avg * (VREF * 1000.0) / 1024.0;
  float temperature = voltage_mV / 10.0;  // LM35: 10 mV / °C
  Serial.print("Temperature:");
  Serial.print(temperature, 2);
  Serial.print(char(0xB0));//remove this line when programming Arduino Uno
  Serial.println("C");  // Then, replace "C" with "°C"
  lcd.setCursor(2, 0);
  lcd.print("Temperature ");
  lcd.setCursor(4, 1);
  lcd.print(temperature);
  lcd.print((char)223);
  lcd.print("C");
  delay(500);
}
// ===== ADC Functions =====
inline void startADC() {
  ADCSRA |= (1 << ADSC);}
inline void waitADC() {
  while (ADCSRA & (1 << ADSC));}
uint16_t readADC() {
  startADC();
  waitADC();
  return ADC;  // ADCL read first automatically
}
/*Sketch uses 4856 bytes (15%) of program storage space. Maximum is 32256 bytes.
```

Global variables use 264 bytes (12%) of dynamic memory, leaving 1784 bytes for local variables. Maximum is 2048 bytes.*/

3.8 Building a Thermistor-Based Temperature Sensor

A thermistor is a variable resistor whose resistance changes with the temperature. A change in resistance means a change in voltages. There are basically two types of thermistors: negative temperature coefficient (NTC) thermistors and positive temperature coefficient (PTC) thermistors. In negative temperature coefficient thermistors, resistance decreases as temperature increases. In positive temperature coefficient thermistors, resistance increases as temperature increases.

To interface a thermistor with an Arduino, a voltage divider circuit is required. The fixed resistor used in the divider should have a resistance value approximately equal to that of the thermistor at the expected operating temperature. Connect the thermistor and the known resistor in series. Then, connect one end of the thermistor to +5 V and the other end of the fixed resistor to ground (Fig.3.6). The output voltage can be measured from the junction between the thermistor and the resistor, which will vary with temperature.

This temperature measurement circuit can also be inverted, meaning the NTC thermistor is connected to ground and the fixed resistor to +5 V. In this configuration, the voltage at the junction still changes with temperature, but the thermistor resistance calculation formula must be adjusted accordingly.

3.8.1 Calculating Temperature

There are various formulas to calculate temperature values from the resistance of the Thermistor. The Steinhart-Hart equation is one of the most accurate ways to calculate the temperature from an NTC thermistor's resistance. The equation is as follows [18]:

$$\frac{1}{T} = A + B\ln(R) + C(\ln(R))^3 \qquad (3.2)$$

where, T is the temperature in Kelvins. R is the resistance of the thermistor at T. A, B and C are the Steinhart–Hart coefficients. To solve this equation, the values of the three constants A, B, and C must be known. These constants vary depending on the type and model of the thermistor, as well as the temperature range of interest. As can be seen from Fig.3.6, the thermistor resistance can be calculated as follows:

$$V_0 = {(R1 \times V)}/{(R1 + R2)} \qquad (3.3)$$

with R1=10kΩ

and

$$ADC_{value} = V_0 \times {1024}/{V_{ref}} \qquad (3.4)$$

and V_{ref}= V. Then:

$${(V \times R1)}/{(R1 + R2)} = ADC_{value} \times ({V}/{1024}) \qquad (3.5)$$

which gives:

$$R2 = R1 \times (\frac{1024}{ADC_{value}} - 1) \qquad (3.6)$$

Example:

This code uses the Steinhart–Hart equation to calculate the temperature based on the circuit shown in Fig.3.7 [19].

```
int thermistorPin = A0;
int Vo;
float R1 =10000;
float logR2, R2, tKelvin, tCelsius, tFahrenheit;
//steinhart-hart coeficients for a 10kΩ thermistor
float c1 = 0.001129148, c2 = 0.000234125, c3 = 0.0000000876741;
void setup(){
```

```
Serial.begin(9600);}

void loop(){
  Vo =analogRead(thermistorPin);
  R2 = R1 *(1023.0/(float)Vo -1.0); // resistance of the Thermistor
  logR2 =log(R2);
  tKelvin =(1.0/(c1 + c2 * logR2 + c3 * logR2 * logR2 * logR2));
tCelsius= tKelvin -273.15;
tFahrenheit=(tCelsius *9.0)/5.0+32.0;
Serial.print("Temperature: ");
Serial.print(tFahrenheit);
Serial.print((char)0xB0);
Serial.print("F; ");
Serial.print(tCelsius);
Serial.print((char)0xB0);
Serial.println("C");
delay(500);
}
```

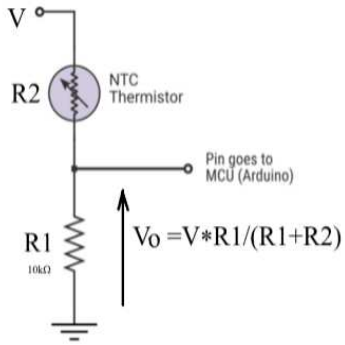


Fig.3.6: Temperature measurement circuit

```
/* Sketch uses 4146 bytes (12%) of program storage space. Maximum is 32256 bytes.
Global variables use 228 bytes (11%) of dynamic memory, leaving 1820 bytes for local variables. Maximum is 2048 bytes.*/
```

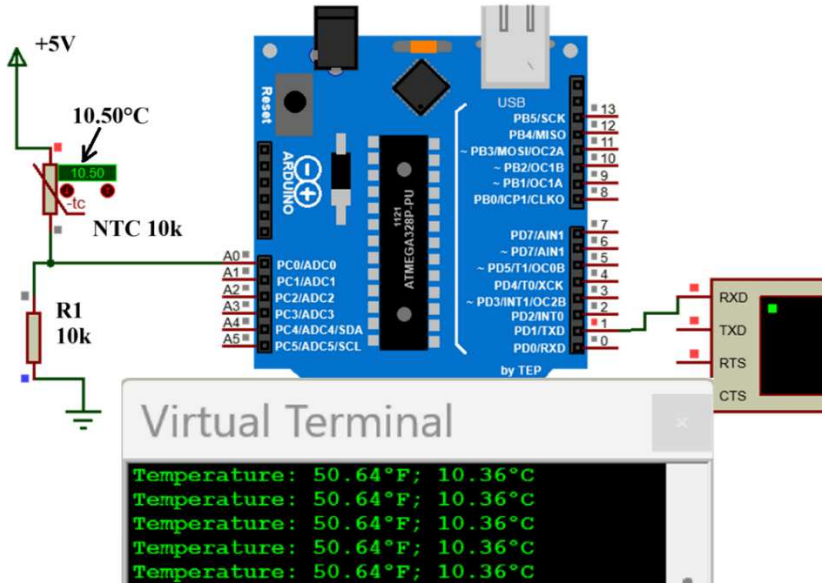


Fig.3.7: Arduino-based Temperature Measurement Circuit Using an NTC Thermistor

For most practical purposes, a simplified version of the Steinhart-Hart equation is used, which only requires the B coefficient (beta value) of the thermistor:

$$\frac{1}{T} = \frac{1}{T_0} + \frac{1}{B} \times \ln\left(\frac{R}{R_0}\right) \qquad (3.7)$$

where:

- T is the temperature in Kelvin.
- T_0 is the nominal temperature (usually 25°C or 298.15K).
- R is the measured resistance of the NTC thermistor.
- R_0 is the nominal resistance at T_0 (e.g., 10kΩ at 25°C).
- B is the beta coefficient of the thermistor (from the datasheet).

Example:
This Arduino code reads the voltage of an NTC thermistor via a voltage divider, calculates the temperature using the simplified version of the Steinhart–Hart equation described above, and prints the result in Celsius to the Serial Monitor every second (Fig.3.8).

```
// Define the analog pin where the NTC is connected
const int ntcPin = A0;
// Define the nominal resistance of the NTC thermistor (10kΩ)
const int nominalResistance = 10000;
// Define the nominal temperature (usually 25°C)
const float nominalTemp = 25.0;
// Define the beta coefficient of the NTC thermistor (check datasheet)
const int betaCoefficient = 3950;
// Define the value of the series resistor (10kΩ)
const int seriesResistor = 10000;
void setup() {
  // Start the Serial communication
  Serial.begin(9600);
}
void loop() {
  // Read the analog value from the NTC thermistor
  int adcValue = analogRead(ntcPin);
  // Convert the analog value to voltage
  float voltage = adcValue * (5.0 / 1024.0);
  // Calculate the resistance of the NTC thermistor
  float ntcResistance = (5.0 * seriesResistor) / voltage - seriesResistor;
  // Calculate the temperature using the Steinhart-Hart equation
  float steinhart;
  steinhart = ntcResistance / nominalResistance;      // (R/Ro)
  steinhart = log(steinhart);                         // ln(R/Ro)
  steinhart /= betaCoefficient;                       // 1/B * ln(R/Ro)
  steinhart += 1.0 / (nominalTemp + 273.15);          // + (1/To)
  steinhart = 1.0 / steinhart;                        // Invert
  steinhart -= 273.15;                                // Convert to Celsius

  // Print the temperature to the Serial Monitor
  Serial.print("Temperature: ");
  Serial.print(steinhart);
  Serial.print((char)0xB0);
  Serial.println("C");
  // Wait for a second before the next reading
  delay(1000);
}
/*Sketch uses 3868 bytes (11%) of program storage space. Maximum is 32256 bytes.
Global variables use 216 bytes (10%) of dynamic memory, leaving 1832 bytes for local variables.
Maximum is 2048 bytes.*/
```

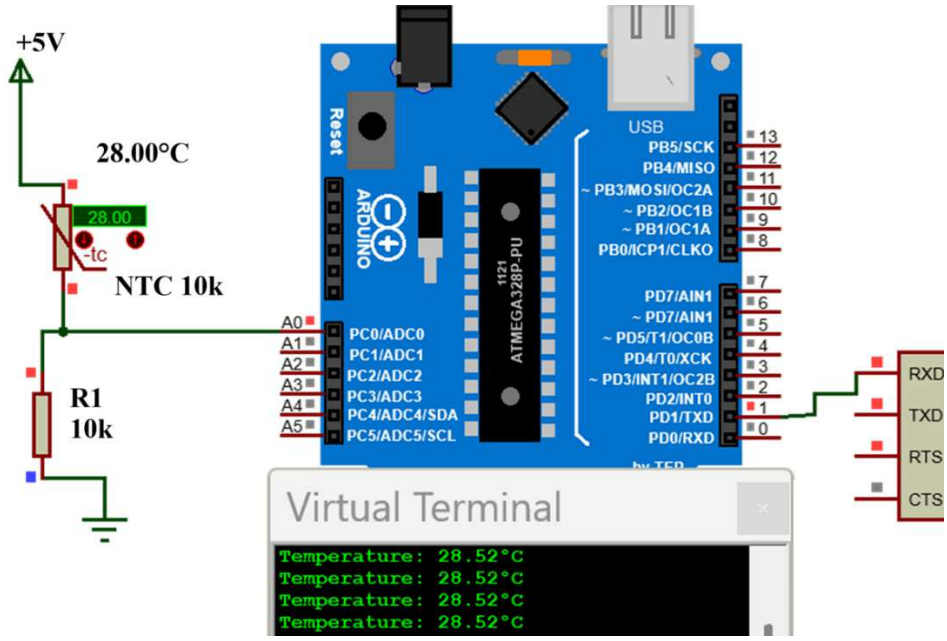


Fig.3.8: Simulation Result

3.9 Digital Temperature Sensor (DS18B20)

The Arduino code below is reading temperature data from a DS18B20 temperature sensor and writing this data to a dual-digit 7-segment display via a 74HC595 shift register. The DS18B20 is operating on pin 4 utilizing the OneWire protocol, and its temperatures readings are obtained and printed over the Serial Monitor (Fig.3.9) [20]. The temperature value is split into its tens and units digits, which are then displayed one at a time by setting the appropriate segment patterns in an array (digitSegments[]). These patterns are shifted out to the 74HC595 using digital pins 8 (DS), 9 (ST_CP), and 10 (SH_CP), which control the serial data line, latch, and clock, respectively.

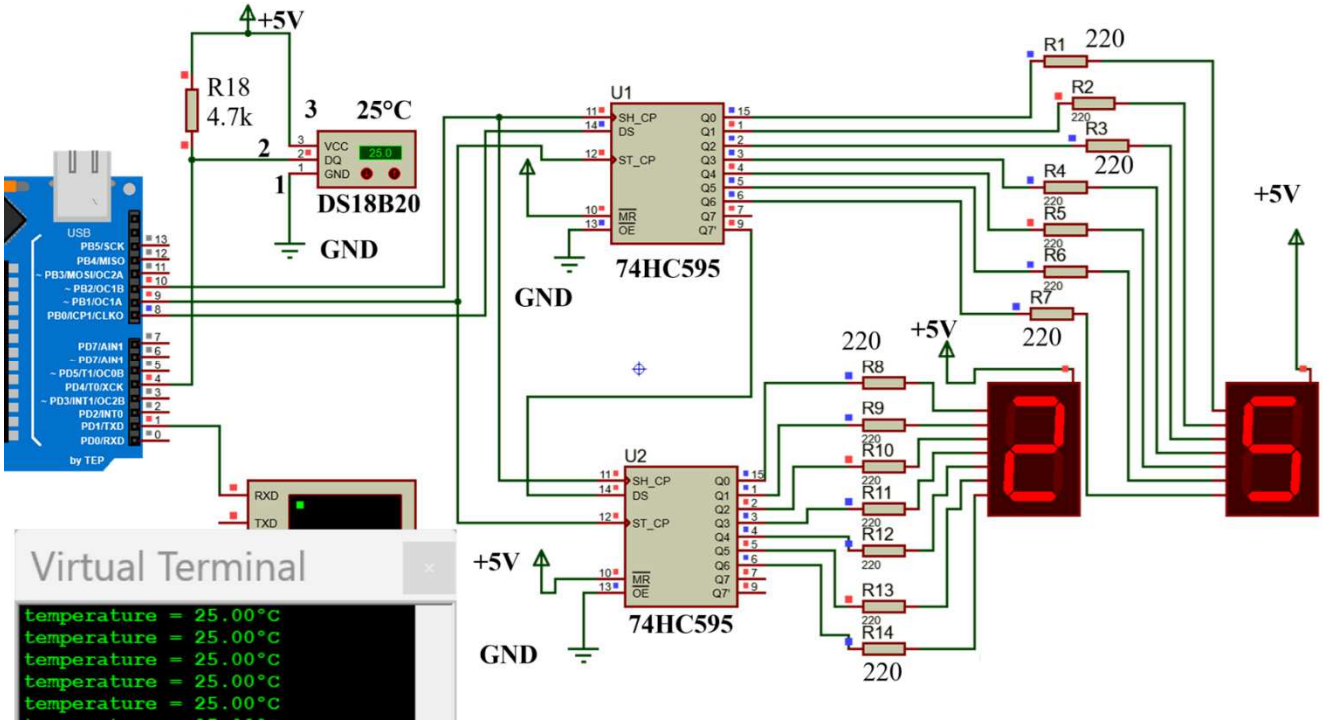


Fig.3.9: Temperature Display System Using DS18B20 and 74HC595 Shift Register

```
#include <OneWire.h>
#include <DallasTemperature.h>
// 74HC595 shift-register pins
const int DS_PIN = 8;     // Serial data input (DS)
const int LATCH_PIN = 9;  // Storage register clock (ST_CP)
const int CLOCK_PIN = 10; // Shift register clock (SH_CP)
// OneWire bus for DS18B20 sensor
#define ONE_WIRE_BUS 4   // DS18B20 data pin on Arduino digital pin 4
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);
// Segment patterns for digits 0–9 (assuming common anode)
// Format: a b c d e f g dp (bit 7 to bit 0)
const byte digitSegments[10] = {
  0b11000000, // 0
  0b11111001, // 1
  0b10100100, // 2
  0b10110000, // 3
  0b10011001, // 4
  0b10010010, // 5
  0b10000010, // 6
  0b11111000, // 7
  0b10000000, // 8
  0b10010000  };  // 9

void setup() {  // Configure shift-register control pins
  pinMode(DS_PIN, OUTPUT);
  pinMode(LATCH_PIN, OUTPUT);
  pinMode(CLOCK_PIN, OUTPUT);
  sensors.begin();  // Start DS18B20 temperature sensor
  Serial.begin(57600);  // Serial communication baud speed
}

void loop() {
  sensors.requestTemperatures();  // Request temperature conversion from DS18B20
  delay(100);
  float temperatureC = sensors.getTempCByIndex(0); // Read temperature in °C from the first sensor
  // Print temperature to serial
  Serial.print("temperature = ");
  Serial.print(temperatureC);
  Serial.print((char)0xB0);
  Serial.println("C");
  // Extract tens and units digits  Example: 27°C -> tens = 2, units = 7
  int tens = int(temperatureC) / 10;
  int units = int(temperatureC) % 10;
  // Display both digits (alternating)
  displayDigit(tens);
  displayDigit(units);
  }
void displayDigit(int digit) {// Sends a digit (0–9) to the seven-segment display
  if (digit < 0 || digit > 9) return;
```

```
  byte segments = digitSegments[digit];
  digitalWrite(LATCH_PIN, LOW);  // Prepare shift register
  for (int i = 7; i >= 0; i--) {              //Shift out 8 bits MSB → LSB
    digitalWrite(CLOCK_PIN, LOW);  // Clock low
    digitalWrite(DS_PIN, (segments >> i) & 0x01);  // Output current bit
    digitalWrite(CLOCK_PIN, HIGH);    // Clock high → bit latched
  }
  digitalWrite(LATCH_PIN, HIGH);      // Transfer shifted bits to output pins
}
/*Sketch uses 5718 bytes (17%) of program storage space. Maximum is 32256 bytes.
Global variables use 269 bytes (13%) of dynamic memory, leaving 1779 bytes for local variables. Maximum is 2048 bytes.*/
```

3.10 Applying a Simple Kalman Filter to Temperature Measurement

The following Arduino code reads temperature data from an analog temperature sensor (LM35) connected to A0, applies a simple Kalman filter to smooth the data, and then displays both the raw and filtered temperature values on the serial monitor and a 16x2 LCD (Fig.3.10).

In the Arduino code, a Kalman filter is applied to smooth the temperature readings from the LM35 sensor, which outputs an analog voltage proportional to temperature [21]. The raw sensor data is first averaged over 50 samples and converted to temperature in °C using the equation:

$$\text{Temperature} = \left(\frac{ADCvalue}{50.0}\right) \times \left(\frac{1.1}{1024.0}\right) \times 100.0 \qquad (3.8)$$

where 1.1 V is the internal analog reference voltage and 1024 is the ADC resolution. The Kalman filter then processes this value using a series of update equations to estimate a more accurate and stable temperature. It starts by predicting the error covariance:

$$\text{Pc} = P + Q \qquad (3.9)$$

where P is the previous estimate error and Q is the process noise variance (varProcess). Next, the Kalman gain is calculated:

$$\text{G} = \frac{Pc}{Pc+R} \qquad (3.10)$$

where R is the measurement noise variance (varVolt). This gain controls how much the new measurement influences the estimate. The estimate error is updated with:

$$\text{P} = (1 - G) \times Pc \qquad (3.11)$$

The predicted state Xp and predicted measurement Zp are both set to the previous estimate Xe. The final updated estimate is computed as:

$$\text{Xe} = G \times (Z - Zp) + Xp \qquad (3.12)$$

where Z is the new raw temperature measurement. The result Xe is the filtered temperature, which reduces noise and provides a smoother reading compared to the raw data.

As shown in Fig.3.11, the practical setup corresponds to the Arduino code that reads and filters temperature data using a Kalman filter, and then displays the result on both the Serial Monitor (Serial Plotter) and the 16x2 LCD.

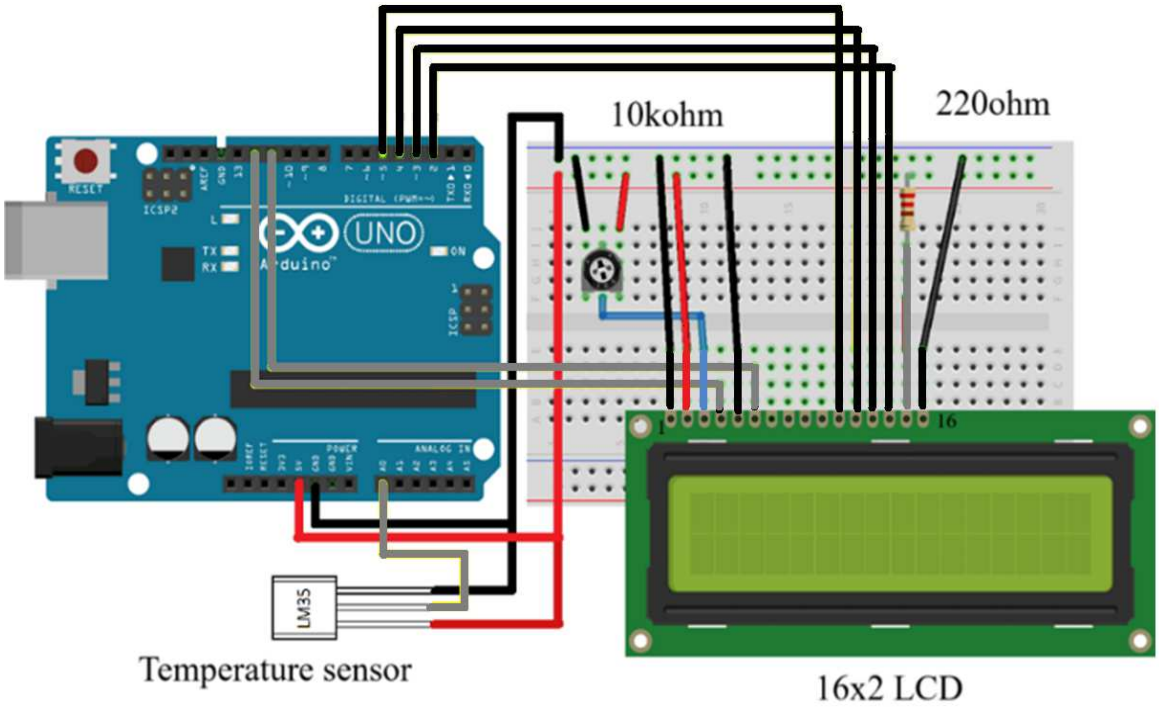


Fig.3.10: Schematic Diagram of a Simple Temperature Monitoring System

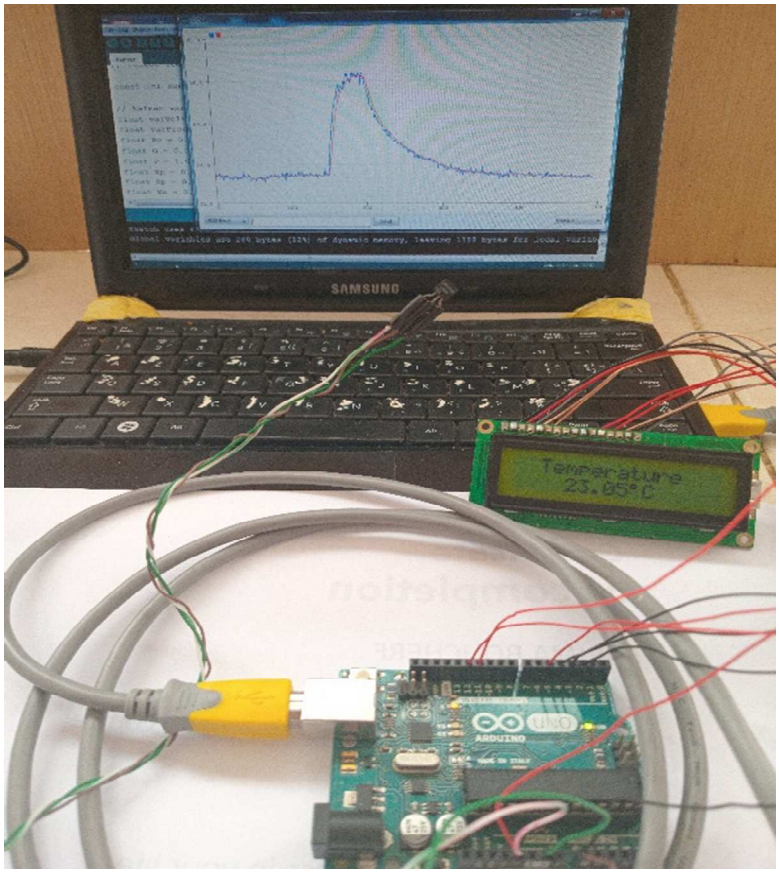


Fig.3.11: Practical Setup

Fig.3.12 shows both curves: the temperature estimated by the Kalman filter (discontinuous curve) and the measured temperature. The peak in the temperature curve corresponds to the sensor being touched with fingers, causing a brief rise in temperature.

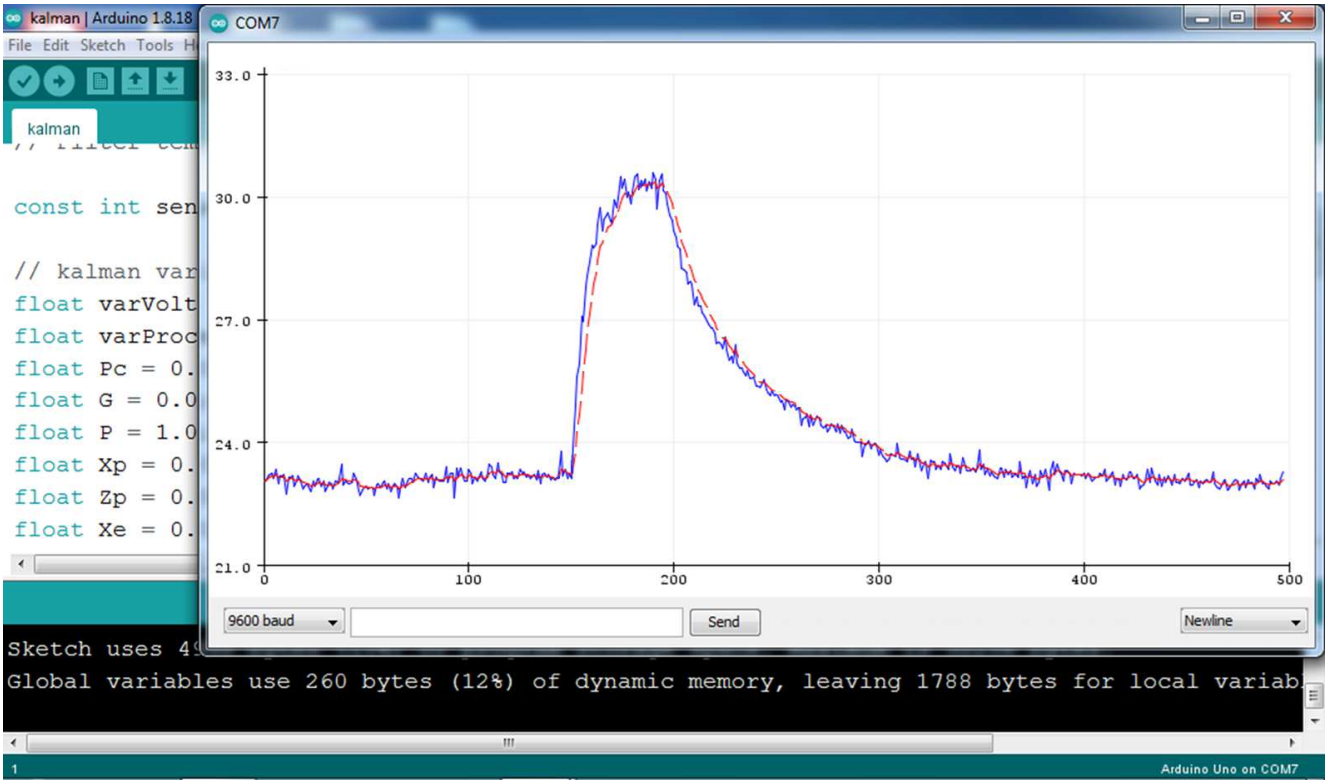


Fig.3.12: Measured and Estimated Temperature Curves

```
#include <LiquidCrystal.h> //LCD Library
// initialize the library by associating any needed LCD interface pin
// with the arduino pin number it is connected to
const int rs = 12, en = 11, d4 = 5, d5 = 4, d6 = 3, d7 = 2;
LiquidCrystal lcd(rs, en, d4, d5, d6, d7);
const int sensorPin = A0;  // named constant for the pin the sensor is connected to
// Filter temperature sensor readings using the Kalman process
// kalman variables
float varVolt = 1.12184278324081E-05;  //variance determined using excel and reading
                                       //samples of raw sensor data
float varProcess = 1e-6;
float Pc = 0.0;
float G = 0.0;
float P = 1.0;
float Xp = 0.0;
float Zp = 0.0;
float Xe = 0.0;
void setup() {
  // open a serial connection to display values
  Serial.begin(9600);
  analogReference(INTERNAL); //Enable ATmega328P Internal Analog Ref of 1.1V
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);}
```

```
void loop() {
  unsigned long ADCvalue=0;
  for(byte i=0;i<50;i++) ADCvalue = ADCvalue + analogRead(sensorPin);
  float temp = ((float(ADCvalue)/50.0) * 1.1 / 1024.0) *100.0; //convert the ADC reading to temperature
  // kalman process
  Pc = P + varProcess;
  G = Pc/(Pc + varVolt);    // kalman gain
  P = (1-G)*Pc;
  Xp = Xe;
  Zp = Xp;
  Xe = G*(temp-Zp)+Xp;   // the kalman estimate of the sensor voltage
  Serial.print(temp);
  Serial.print(",");
  Serial.println(Xe);
  //set the cursor to column 2, line 0
  lcd.setCursor(2, 0);
  // Print a message to the LCD.
  lcd.print("Temperature");
  // set the cursor to column 4, line 1
  // (note: line 1 is the second row, since counting begins with 0):
  lcd.setCursor(4, 1);
  lcd.print(temp);
  lcd.print(char(0xDF)); //used to print the degree symbol (°)
  lcd.print("C");
  delay(1000);
}
/* Sketch uses 4988 bytes (15%) of program storage space. Maximum is 32256 bytes.
Global variables use 260 bytes (12%) of dynamic memory, leaving 1788 bytes for local variables.
Maximum is 2048 bytes.*/
```

3.11 Conclusion

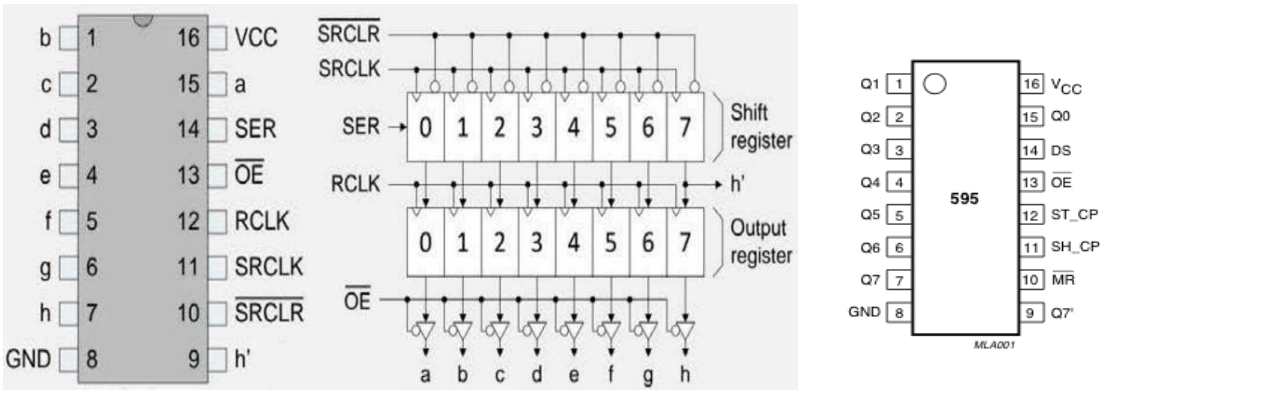
This chapter presented the primary digital and analog I/O operations used with Arduino. It also demonstrated several temperature-sensing methods using different sensors. The next chapter will focus on the use of a shift register, a port expander, and interrupts, explaining how they can be applied in practice.

4.1 Introduction

This chapter explores techniques for expanding the ATmega328P microcontroller's I/O capabilities and improving its responsiveness using shift registers, port expanders, and interrupts. It also covers controlling multiple outputs with minimal pins, interfacing devices through I²C, and handling real-time events efficiently.

4.2 Serial to Parallel Shifting-Out with a 74HC595

The 74HC595 controls eight different output pins using only three input pins (Table 4.1). When more than eight I/O pins are required, multiple shift registers can be daisy-chained to generate a larger number of outputs (Fig.4.1) [15][16]. This is achieved through a technique known as bit-shifting.



(a) Package and block diagram **(b) 74HC595 pinout (Proteus Isis)**

Fig.4.1: The Pinout and Internal Structure of a 74HC595 Serial-to-Parallel Shift Register.

Table 4.1: Pin functions of the 74LS595

PINS 1-7, 15	Q0 - Q7	Output Pins
PIN 8	GND	Ground, Vss
PIN 9	Q7'	Serial Out
PIN 10	MR	Master Reclear, active low
PIN 11	SH_CP	Shift register clock pin
PIN 12	ST_CP	Storage register clock pin (latch pin)
PIN 13	OE	Output enable, active low
PIN 14	DS	Serial data input
PIN 16	Vcc	Positive supply voltage

The shift register has eight internal memory positions, each of which is capable of holding a binary value. To set each of these values ON or OFF, we feed in the data using the 'Data' and 'Clock' pins of the chip. The clock pin needs to receive eight pulses. At the time of each pulse, if the data pin is high, then a 1 gets pushed into the shift register. Otherwise, it is a 0. When eight clock pulses are exceeded, the latch pin enables these eight values to be loaded into the latch register to be accessed on the output pins (Fig.4.2). The OE (Output Enable) pin also controls whether the outputs are active or in a high-impedance state (disabled).

4.3 Controlling Seven-Segment Displays with Shift Registers

The following code is for an Arduino Uno interfaced with a 74HC595 shift registers to drive 7-segment common anode displays [16]. The Latter show the temperature read from analog pin A0 using the internal voltage reference. This process repeats every 3 seconds to update the displays with the latest temperature reading. The practical setup, shown in Fig.4.3, uses an LM35 temperature sensor to display a temperature of 24.7 °C. The main difference between the simulation and the practical setup is the placement of the decimal point, which appears on the third digit in the practical setup (Fig.4.4), and its LED is connected to VCC.

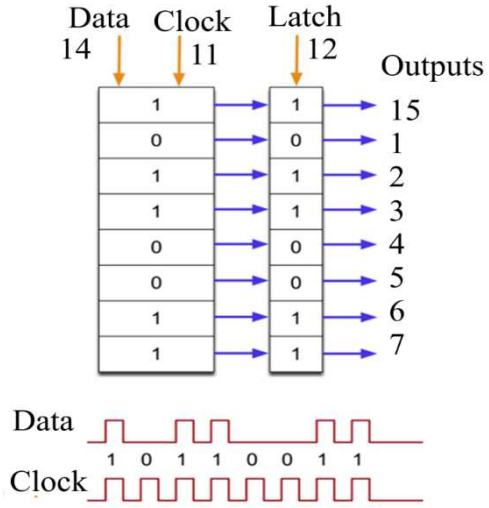


Fig.4.2: Operation of the 74HC595 Shift Register

```
// ATmega328P (Arduino Uno) -> Shift register 74HC595
// PB0 (pin 8) -> DS (data input)
// PB2 (pin 10) -> ST_CP (latch pin)
// PB3 (pin 11) -> SH_CP (clock pin)
unsigned long prev, now, T = 3000; // T = display period (3 seconds) for 7-segment common-anode
                                   //display
byte i = 0, j = 0, l = 0, out1;
float temp;
void setup() {
   // Set PB0, PB2, PB3 as outputs for shift register control
   DDRB |= (1 << PB0);
   DDRB |= (1 << PB2);
   DDRB |= (1 << PB3);
   // Enabling ADC
   ADCSRA |= (1 << ADEN);
   // Initialize display with 00
   numberdisp(j);
   numberdisp(i);
   prev = millis();   // Start timing
}

void writePins()
{    // Latch low before shifting data
```

```
    PORTB &= ~(1 << PB2);
    // Shift out 8 bits, MSB first
    for (int i = 7; i >= 0; i--)
    {   PORTB &= ~(1 << PB3);        // Clock low
        // Check if current bit is 1, send it to DS
        if (out1 & (1 << i))
            PORTB |= (1 << PB0);     // DS HIGH
        // else DS LOW (already done below)
        PORTB |= (1 << PB3);         // Rising edge clocks bit into 74HC595
        PORTB &= ~(1 << PB0);        // Reset data line
    }
    PORTB |= (1 << PB2);             // Latch HIGH to output stored data
}
void loop() {
    temp = 0.00;
    now = millis();
    // Every 3 seconds, take ADC reading and update display
    if ((now - prev) >= T)
    {   // Take 20 readings and average them
        for (int k = 1; k <= 20; k++)
            temp = temp + ADConversion() / 20.0;
        // Valid temperature range 1.0 – 99.0
        if ((temp >= 1.0) && (temp <= 99.0))
        {  j = temp / 10.0;      // Tens digit
            numberdisp(j);
            i = temp - j * 10;    // Units digit
            numberdisp(i);
            l = (temp - j * 10 - i) * 10.0;  // Decimal digit
            numberdisp(l);
        }
        else
        {  // Display "Err" for out-of-range values
            numberdisp(10);  // E
            numberdisp(11);  // r
            numberdisp(11);  // r
        }
        prev = now; // Reset timer
    }
}
void numberdisp(byte number)
{  // Segment patterns for common-anode 7-seg display (active LOW)
    if (number == 0)  out1 = 0x40;

    else if (number == 1)  out1 = 0x79;
    else if (number == 2)  out1 = 0x24;
```

```
    else if (number == 3)  out1 = 0x30;
    else if (number == 4)  out1 = 0x19;
    else if (number == 5)  out1 = 0x12;
    else if (number == 6)  out1 = 0x02;
    else if (number == 7)  out1 = 0x78;
    else if (number == 8)  out1 = 0x00;
    else if (number == 9)  out1 = 0x10;
    else if (number == 10) out1 = 0x86; // "E"
    else if (number == 11) out1 = 0xAF; // "r"
    writePins(); // Send pattern to shift register
}
float ADConversion()
{    // Select ADC channel 0, use internal reference, left default for others
       ADMUX |= (1 << REFS1) | (1 << REFS0);   // REFS1:0 = 11, MUX bits = 0
    // Enable ADC, start conversion, and set prescaler = 128
    ADCSRA |= (1 << ADSC) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
    // Wait for conversion to complete (ADSC bit becomes 0)
    while (ADCSRA & (1 << ADSC));
    // Combine low + high bytes and convert to temperature
    float val = (ADCL | (ADCH << 8)) * 110.0 / 1024.0;
    return val;
}
/* Sketch uses 2240 bytes (6%) of program storage space. Maximum is 32256 bytes.
Global variables use 24 bytes (1%) of dynamic memory, leaving 2024 bytes for local variables.
Maximum is 2048 bytes. */
```

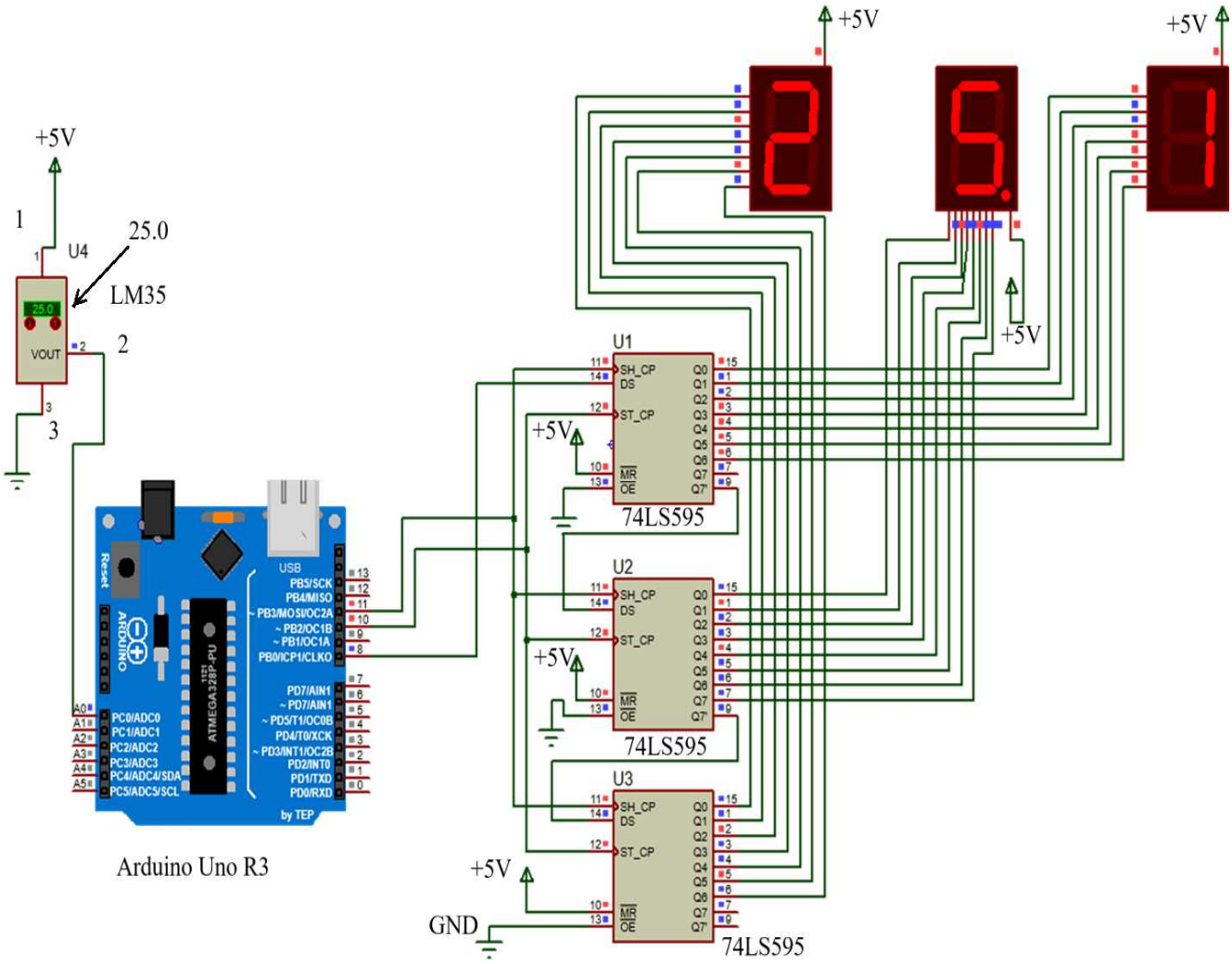


Fig.4.3: Temperature Monitoring System

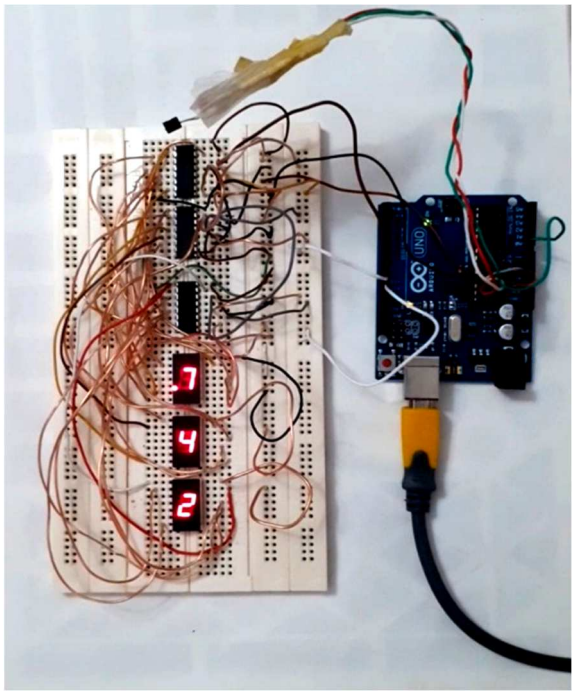


Fig.4.4: Practical Setup

4.4 Using a Port Expander (PCF8574)

The PCF8574 is an I²C-based 8-bit GPIO expander that allows microcontrollers such as the ATmega328P or ESP8266 to gain eight additional digital I/O pins (P0–P7) using only two wires (SDA and SCL). It is commonly employed in applications such as LCD modules, button matrices, and relay controls. The pinout and pin definitions are provided in Fig.4.5 and Table 4.2.

Table 4.2: Pinout Summary

Pin	Function	Description
A0–A2	Address Select	Sets I²C address
P0–P7	GPIO Pins	8-bit I/O (Input/Output)
INT	Interrupt Output	Triggers on input change (if enabled)
SCL	I²C Clock	Serial clock line
SDA	I²C Data	Serial data line
VCC	Power Supply	+5V or +3.3V
GND	Ground	0V reference

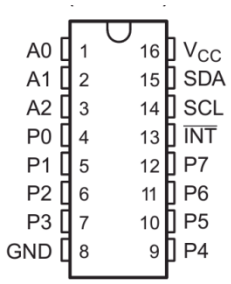


Fig.4.5: PCF8574 pinout

The PCF8574 does not have internal pull-up resistors on the I²C lines, so external pull-ups (typically 4.7kΩ–10kΩ) are needed for SDA and SCL. The GPIO pins are quasi-bidirectional, meaning they can function as input or output without explicit direction setting, but may need external pull-ups when used as inputs or to ensure a strong HIGH state. Fig.4.6 illustrates two different configurations for driving an LED with the PCF8574, showing the differences between sinking 1 mA and sourcing 25 mA of current.

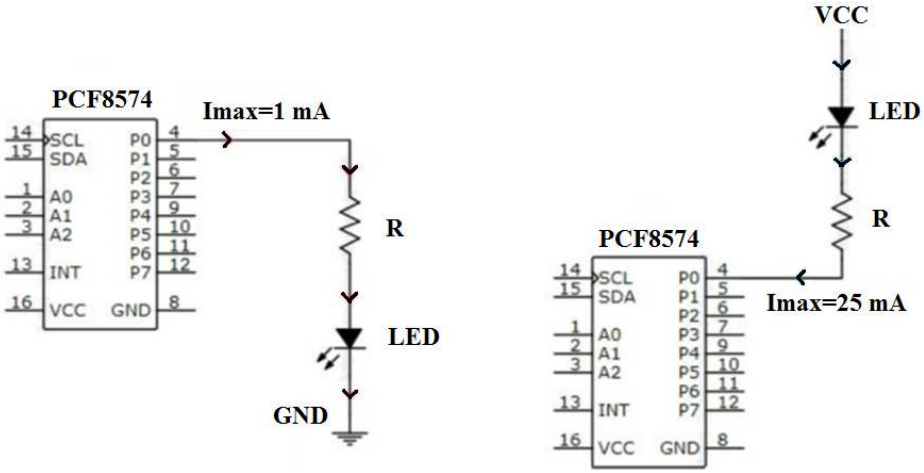
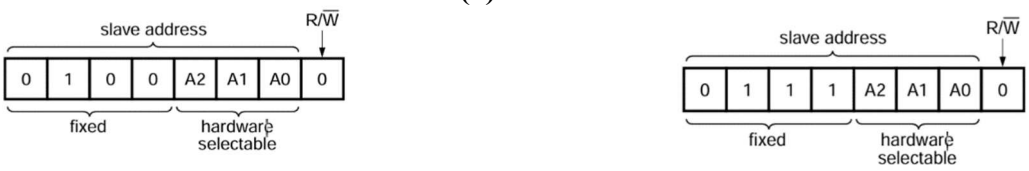


Fig.4.6: Comparison of LED Drive Modes on the PCF8574

While the I²C standard supports 7-bit addressing (Fig. 4.7.a), the PCF8574 uses only three bits of this address space, limiting its possible addresses to the range 0x20–0x27 (Table 4.3) [22]. The slave address formats for the PCF8574 and PCF8574A are shown in Fig. 4.7.b and Fig. 4.7.c, respectively. In these formats, A2, A1, and A0 are hardware-configurable address bits, while the R/W bit indicates the operation type: read (1) or write (0).

Start	Device Address	R/W	ACK (0)	Data	ACK (0)	Data	ACK (0)	Stop
1 Bit	7 or 10 Bits	1 Bit	1 Bit	8 Bits	1 Bit	8 Bits	1 Bit	1 Bit
From master			From slave	From master	From slave	From master	From slave	From master

(a)



(b) PCF8574 Address Format **(c) PCF8574A Address Format**

Fig.4.7: I²C Communication Format

Table 4.3: Address values for PCF8574/A

A2	A1	A0	PCF8574 Address	PCF8574A Address
L	L	L	0x20	0x38
L	L	H	0x21	0x39
L	H	L	0x22	0x3A
L	H	H	0x23	0x3B
H	L	L	0x24	0x3C
H	L	H	0x25	0x3D
H	H	L	0x26	0x3E
H	H	H	0x27	0x3F

4.4.1 Arduino Control of a Seven-Segment Display with PCF8574

The circuit of Fig.4.8 allows an Arduino Uno to control a seven-segment LED display using a PCF8574 I²C I/O expander, reducing the number of pins needed on the Arduino. The SDA and SCL lines from the Arduino connect to the PCF8574, with 10 kΩ pull-up resistors ensuring proper I²C communication. The PCF8574 outputs (P0 to P7) drive the individual LED segments of the seven-segment display through 220 Ω current-limiting resistors. When the Arduino sends commands over I²C, the PCF8574 sets the required pins high or low, lighting the appropriate segments so the display can show digits or characters.

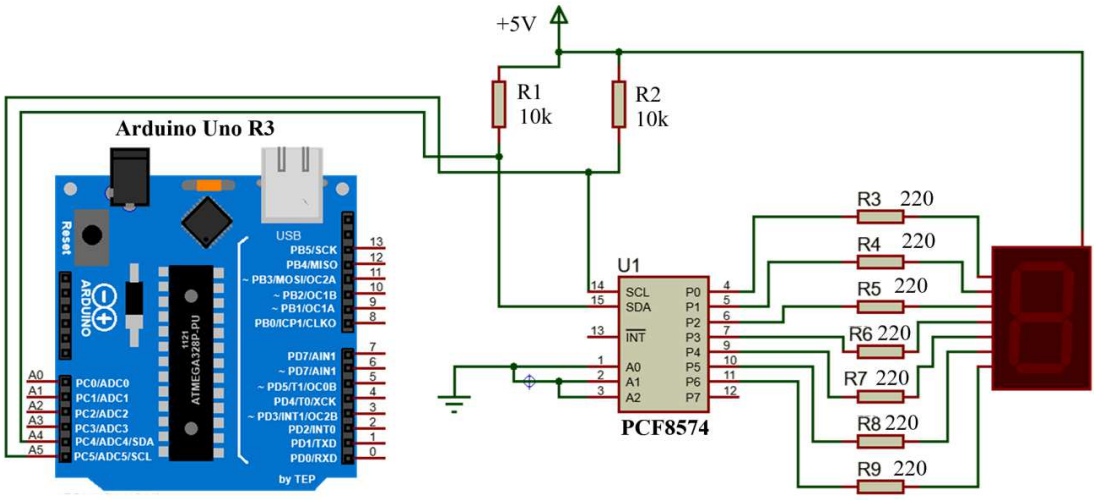


Fig.4.8: Circuit Diagram of Arduino, PCF8574, and a 7-Segment Display

The below sketch code that drives a seven-segment display via PCF8574. It cycles through digits 0-9, displaying each for 500ms. The displnumber() function converts numbers to seven-segment bit patterns (common anode) and sends them to the PCF8574 (I²C address 0x20, A0=0 A1=0 A2=0). Each bit (P0 to P7) in number controls a segment (active-low).

```
#include <Wire.h>                 // I²C library for communication with PCF8574 I/O expander
const byte pcf8574_addr = 0x20;     // I²C address of PCF8574 (A0=A1=A2=LOW → 0x20)
void setup() {
  Wire.begin();                   // Initialize I²C bus
}
void loop() {
  byte i;
  // Display numbers 0–9 on the 7-segment using PCF8574
  for (i = 0; i <= 9; i++)
  {    displnumber(i, pcf8574_addr);   // Send digit pattern via I²C
    delay(500);                   // Half-second delay before next number
  }
}
void displnumber(byte number, byte address)
{ // Convert number (0–9) to its 7-segment encoding (common anode)
  // Patterns are active LOW (0 = LED ON)
  if(number == 0) number = 0xC0;
  else if (number == 1) number = 0xF9;
  else if (number == 2) number = 0xA4;
  else if (number == 3) number = 0xB0;
  else if (number == 4) number = 0x99;
  else if (number == 5) number = 0x92;
  else if (number == 6) number = 0x82;
  else if (number == 7) number = 0xF8;
  else if (number == 8) number = 0x80;
  else if (number == 9) number = 0x90;
  // Send byte to PCF8574 over I2C
  Wire.beginTransmission(address);  // Start I²C transmission
  Wire.write(number);               // Write segment byte to the expander
  Wire.endTransmission();           // Finish transmission (PCF8574 outputs are updated)
}
/*Sketch uses 2644 bytes (8%) of program storage space. Maximum is 32256 bytes.
Global variables use 225 bytes (10%) of dynamic memory, leaving 1823 bytes for local variables.
Maximum is 2048 bytes.*/
```

4.4.2 Temperature Measurement with Arduino Uno and I²C LCD

Fig.4.10 shows an Arduino-based temperature-monitoring setup that displays the measured temperature on an I²C LCD screen. The provided sketch reads an LM35 analog temperature sensor connected to pin A0, averages 100 samples every second, converts the result using the internal 1.1V reference, and outputs the calculated temperature to both the serial monitor and a 16×2 I²C LCD. The LCD uses the I²C address 0x27, which is selected by leaving the address header pins A0, A1, and A2

kept open (unconnected to GND), giving the module its default address (Fig.4.9). In the program, the LCD is initialized at this address; it prints the label “Temperature” on the first line and displays the measured temperature in degrees Celsius on the second line. Fig. 4.11 shows the Arduino IDE with the serial monitor continuously displaying temperature readings of approximately 23.4 °C.

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
// Initialize LCD at I2C address 0x27, configured as 16 columns × 2 rows
LiquidCrystal_I2C lcd(0x27, 16, 2);
unsigned long period = 1000;          // Sampling interval = 1 second
unsigned long prev_sample_time;       // Stores the timestamp of the last sample
void setup()
{
  Serial.begin(57600);                // Start serial communication

  lcd.init();                         // Initialize the LCD
  analogReference(INTERNAL);          // Use internal 1.1 V reference for analog readings
  prev_sample_time = millis();        // Record initial time marker
}
void loop()
{
  unsigned long data = 0;             // Variable to accumulate analog samples
  unsigned long actual_sample_time = millis();
  // Check if 1 second has passed since the last reading
  if ((actual_sample_time - prev_sample_time) >= period)
  {
    prev_sample_time = actual_sample_time;  // Update timestamp
    // Take 100 analog samples and sum them
    for (int i = 0; i <= 99; i++)
      data += analogRead(0);          // Read LM35 sensor on A0
    // Convert ADC value to temperature (°C)
    // LM35 outputs 10 mV/°C; ADC uses internal 1.1V reference
    float temp = ((float)data) * 1.1 / 1024.0;
    // Print to Serial Monitor
    Serial.print(temp);
    Serial.println("°C");
    // Display on LCD
    lcd.backlight();                  // Ensure LCD backlight is ON
    lcd.setCursor(3, 0);              // First row, column 3
    lcd.print("Temperature");

    lcd.setCursor(5, 1);              // Second row, column 5
    lcd.print(temp, 1);               // Print temperature with 1 decimal place
    lcd.print((char)223);             // Print degree symbol
```

```
lcd.print("C");
  }
}
```

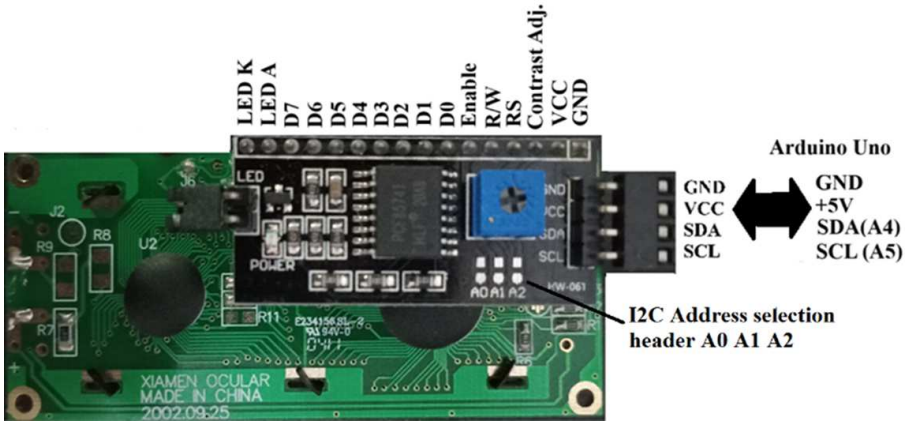


Fig.4.9: Pin Mapping Between the I²C LCD and Arduino Uno

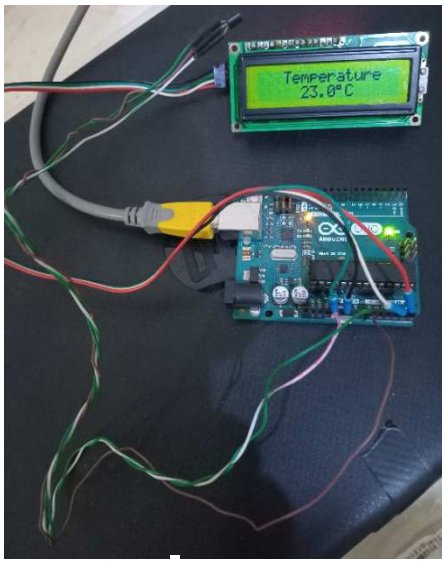


Fig.4.10: Arduino Uno with I²C LCD Showing Temperature Reading

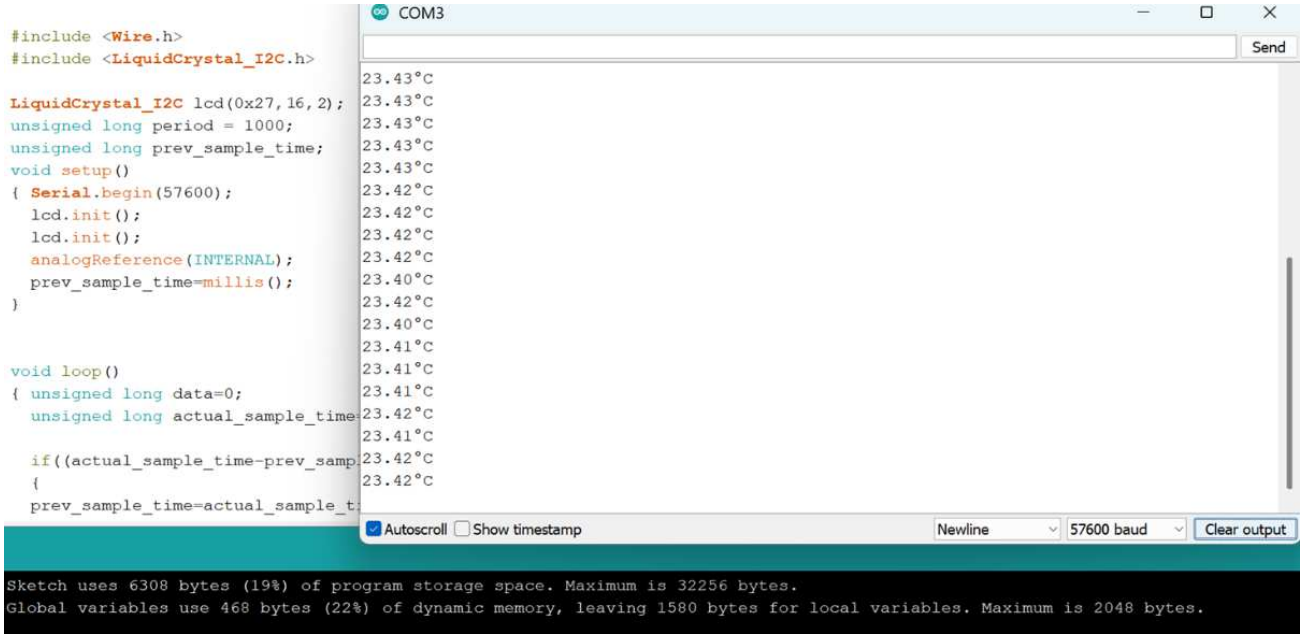


Fig.4.11: Temperature Readings Displayed on the Serial Monitor Window

4.4.3 Register-Level Implementation of a Temperature Measurement System

The previous setup (Fig.4.10) is used for register-level ATmega328P programming to implement I²C LCD control, ADC temperature sampling, UART communication at 57600 bps, and a custom Timer0-based timing system without relying on Arduino high-level libraries [10]. The Timer0 interrupt generates a precise 1ms millis() counter, effectively replacing Arduino's built-in timing functions (the sketch code is provided in annex2). Figure 4.12 shows the serial monitor output, where temperature values are transmitted via UART and updated every second using this custom Timer0 timing routine.

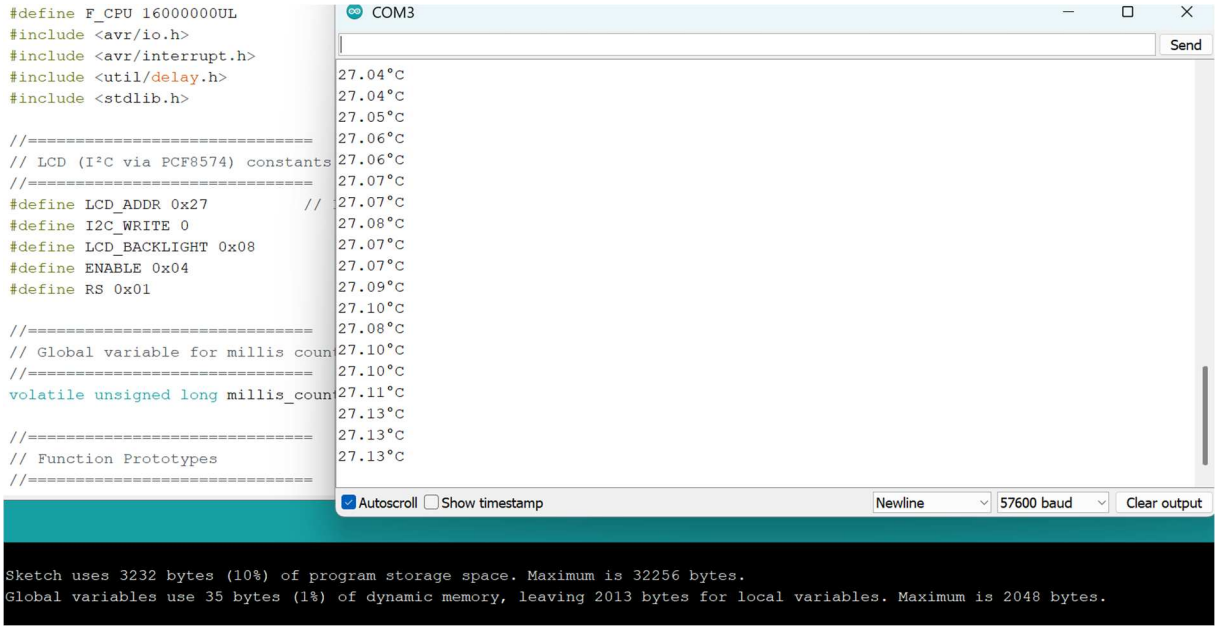


Fig.4.12: Serial Monitor Temperature Readings

As shown in Table 4.4, the register-level implementation is significantly more memory-efficient than the Arduino high-level version. The SRAM usage is drastically reduced because no heavy Arduino libraries (e.g., Wire.h, LiquidCrystal_I2C.h, or the Serial.print() framework) are loaded. Flash consumption is also reduced by nearly 50%, leaving substantially more program space available for future extensions or additional features.

Table 4.4: Memory Usage Comparison

Version	Flash	SRAM
Low-Level Register Version	3232 bytes (10% of 32256 bytes)	35 bytes (1% of 2048 bytes)
Arduino High-Level Version	6308 bytes (19% of 32256 bytes)	468 bytes (22% of 2048 bytes)

4.5 ADC Conversion in Free-Running Mode with Interrupts

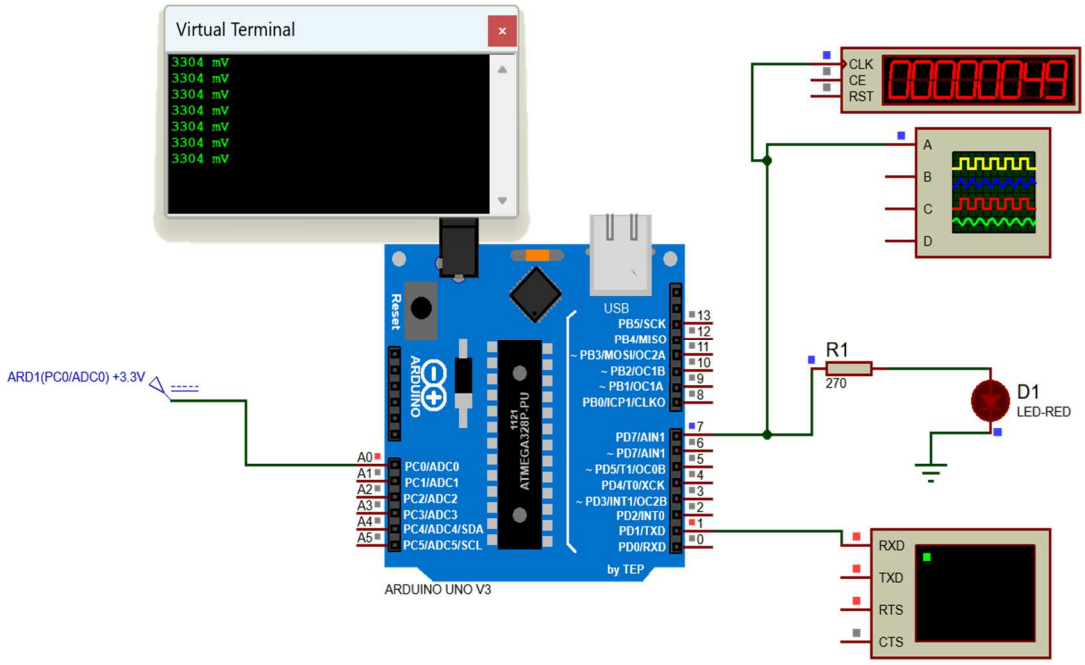
As explained in Chapter 3 (Section 3.5), in free-running mode the ADC automatically begins a new conversion as soon as the previous one finishes [15]. With interrupts enabled, there is no need to repeatedly poll the ADSC bit; instead, an interrupt is generated after each completed conversion. The following example configures the ADC on A0 to operate in free-running mode, accumulates 100 samples, averages them, and processes the result using an interrupt-driven approach. Fig. 4.13 illustrates the Arduino Uno simulation showing ADC voltage readings sent to the virtual terminal along with an LED indicator.

```
#define SAMPLE_COUNT 100U
#define VREF_MV      5000UL   // change if using different Vref (e.g., 5000 mV for AVCC=5V)
```

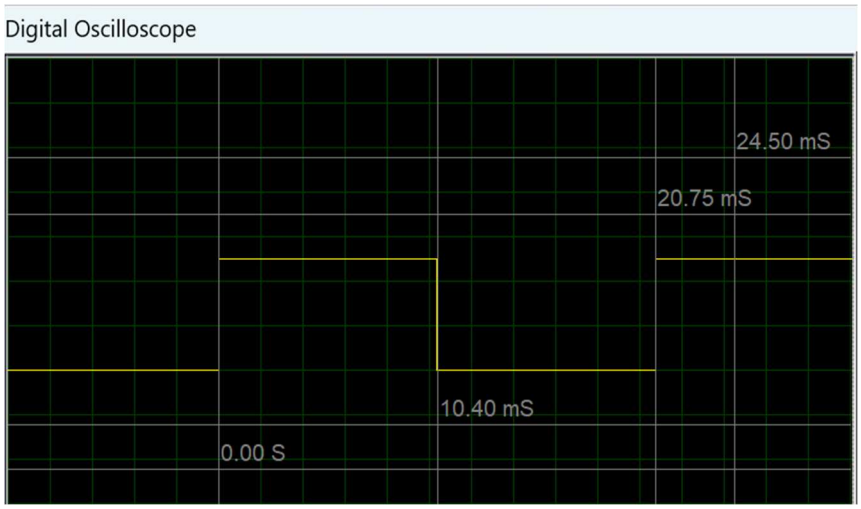
```
volatile uint32_t adc_accumulator = 0;   // accumulate ADC results (must be large enough)
volatile uint16_t adc_sample_index = 0;  // number of samples accumulated so far
                                                    //(0..SAMPLE_COUNT)
volatile uint16_t adc_average = 0;       // holds computed average (0..1023)
volatile bool adc_average_ready = false; // flag set when average is computed
uint8_t counter_index = 0;//loop counter
void ADC_init_free_running_A0(void) {
   // Select reference = AVCC (REFS0=1), right-adjust result (ADLAR=0), MUX = 0 (ADC0)
   ADMUX = (1 << REFS0) | (0 << ADLAR) | (0 << MUX0); // ADMUX low nibble already 0 =>
                                                                  //ADC0
   // ADCSRA:
   // ADEN  = 1 (enable ADC)
   // ADSC  = 1 (start first conversion)
   // ADATE = 1 (auto trigger enable)
   // ADIE  = 1 (interrupt enable)
   // ADPS[2:0] = prescaler => choose 128 for 16MHz->125kHz ADC clock (safe)
   ADCSRA = (1 << ADEN)  |
            (1 << ADSC)  |
            (1 << ADATE) |
            (1 << ADIE)  |
            (1 << ADPS2) |
            (1 << ADPS1) |
            (1 << ADPS0); // prescaler = 128
   // ADCSRB: ADTS[2:0] = 000 => Free running mode
   ADCSRB = 0x00;
   // Note: ADIF will be set after first conversion finishes; with ADIE=1
   sei(); // enable global interrupts by setting the I-bit (bit 7) in the Status Register (SREG)
}
ISR(ADC_vect) {
   // This ISR is called on every ADC conversion completion.
   // Read the 10-bit result (ADCL then ADCH or via ADC register)
   uint16_t val = ADC; // ADC is 16-bit register mapping ADCL/ADCH -> gives 0..1023
   adc_accumulator += (uint32_t)val;
   adc_sample_index++;
   if (adc_sample_index >= SAMPLE_COUNT) {
      // compute average
      adc_average = (uint16_t)(adc_accumulator / SAMPLE_COUNT);
      // reset accumulator and index for the next averaging window
      adc_accumulator = 0;
      adc_sample_index = 0;
      // indicate to main loop that averaged value is available
      adc_average_ready = true;
   }
   // ADIF is handled by hardware when an interrupt is executed; no manual clear required here.
}
```

```
void setup(){
  // (Optional) configure an LED pin to show new average ready (PD7 for example)
    DDRD |= (1 << PD7); // LED on pin D7
   Serial.begin(57600);
   ADC_init_free_running_A0();
   }
void loop() {
        if (adc_average_ready) {
         counter_index++;
         // To safely read multi-byte volatile adc_average we temporarily disable ADC interrupt
         cli(); // disable global interrupts
         uint16_t avg = adc_average;
         adc_average_ready = false;
         sei(); // re-enable interrupts
         // Example: convert average to millivolts assuming Vref = AVCC = 5.000 V
         // mv = avg * VREF_MV / 1023
         uint32_t mv = ((uint32_t)avg * VREF_MV) / 1023UL;

         // toggle LED, update display, send over UART, adjust PWM, etc.
         // Example: toggle LED pin (if configured)
          PORTD ^= (1 << PD7);
          if(counter_index==100){
          counter_index=0;
          Serial.print(mv);
          Serial.println(" mV");
          }
      }
}
```



(a) Arduino-Based Circuit Simulation



(b) LED Applied Signal Waveform

Fig.4.13: Simulation Setup of the ADC Conversion in Free-Running Mode with Interrupts

4.6 Digital Pins with Hardware Interrupts

When using attachInterrupt() function, the first argument specifies the interrupt number associated with a particular digital pin [9][23]. Instead of manually determining this number, it is recommended to use digitalPinToInterrupt(pin), which automatically converts the digital pin into its corresponding interrupt number. For example, if an interrupt is to be triggered from digital pin D3, the correct usage is attachInterrupt(digitalPinToInterrupt(3), ...). On Arduino boards such as Uno, Nano, Mini, and other ATmega328P-based variants, only digital pins D2 and D3 can be used as external interrupt pins.

Syntax: attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)

where:

pin: the Arduino pin number.

ISR: the ISR to call when the interrupt occurs; this function must take no parameters and return nothing.

mode: defines when the interrupt should be triggered. Four constants are predefined as valid values [20]:

LOW: to trigger the interrupt whenever the pin is low,

CHANGE: to trigger the interrupt whenever the pin changes value,

RISING: when the pin goes from low to high,

FALLING: when the pin goes from high to low.

When hardware interrupts are enabled, an application may need to change the trigger mode of an interrupt, for example, switching from RISING to FALLING. To make this change, the interrupt must first be disabled using the detachInterrupt() function. This function takes a single parameter that specifies which interrupt to disable, typically 0 or 1. After the interrupt has been turned off, it can be reconfigured with a new mode by calling attachInterrupt() again with the updated settings.

It should be noted that inside the attached function, delay() won't work, the value returned by millis() will not increment, and serial data received while in the function may be lost. Furthermore, any variables used within the attached function should be declared as volatile.

Generally, an ISR is a special function which should be as short and fast as possible. Furthermore, it cannot have any parameters, and it shouldn't return anything. If a sketch code uses multiple ISRs,

only one can run at a time, other interrupts will be executed after the current one finishes in an order that depends on the priority they have.
It known that millis() function relies on interrupts to count, therefore, it will never increment inside an ISR. Since delay() function requires interrupts to work, it will not work if called inside an ISR. micros() function works initially but will start behaving erratically after 1-2 ms. delayMicroseconds() function does not use any counter, so it will work as normal [9].
Example:
This code toggles an LED (pin D5) when a button (pin D2) is pressed, using interrupts for instant response (Fig.4.14). The button uses the microcontroller's internal pull-up, so pressing it connects to GND (FALLING edge). The LED changes state (ON/OFF) with every button press, even during the delay(500) in loop().

```
const uint8_t btn_pin=2;   // PD2
const uint8_t led_pin=5;   // PD5
void setup() {
//set button pin to be input with pullup
DDRD &=~(1<<btn_pin);
PORTD |=(1<<btn_pin);
//set led pin to be out
DDRD |=(1<<led_pin);
attachInterrupt(digitalPinToInterrupt(btn_pin),blink,FALLING);
}
void loop() {
delay(500);
}
void blink()   //ISR
{
  PORTD ^=(1<<led_pin);
}
/*Sketch uses 818 bytes (2%) of program storage space. Maximum is 32256 bytes.
Global variables use 13 bytes (0%) of dynamic memory, leaving 2035 bytes for local variables.
Maximum is 2048 bytes.*/
```

Example:
This code counts the number of falling edges (like button presses) using an external interrupt on D2 (PD2) (Fig.4.13). After 5 such events, it toggles the LED connected to pin D13 (PB5) and resets the counter to start counting again.

```
volatile uint8_t cnt=0;
void setup() {
// Configure pin PB5 as output
DDRB |=(1<<PORTB5);
// Configure pin PD2 as input
DDRD &=~(1<<PORTD2);
// Enable internal pull-up resistor on PD2
PORTD |=(1<<PORTD2);
```

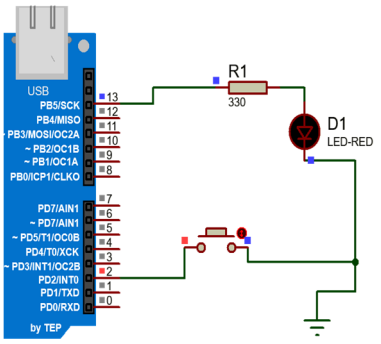


Fig.4.14: Arduino LED Control with Pushbutton Circuit

```
// Configure INT0 to trigger on the falling edge
// ISC01 = 1, ISC00 = 0 → falling edge detection
//EICRA – External Interrupt Control Register A
EICRA |=(1<<ISC01);
EICRA &=~(1<<ISC00);
// Enable external interrupt INT0
//EIMSK – External Interrupt Mask Register
EIMSK |=(1<<INT0);
// Enable global interrupts
// Global interrupts enable (sei();Enable interrupts, cei():Disable interrupts)
sei();
}
void loop() {
  // put your main code here, to run repeatedly:
if(cnt>=5) {PORTB ^=(1<<PORTB5); cnt=0;}
}
ISR(INT0_vect)
{
  cnt++;
}
/*Sketch uses 528 bytes (1%) of program storage space. Maximum is 32256 bytes.
Global variables use 10 bytes (0%) of dynamic memory, leaving 2038 bytes for local variables. Maximum is 2048 bytes.*/
```

4.7 Pin Change Interrupts (PCINT)

The advantages of PCINT (Pin Change Interrupt) interrupts lie in their flexibility: they can be triggered by virtually any digital pin on the Arduino. Unlike INT0 and INT1 interrupts, which are restricted to specific pins and can be configured for particular edge types, PCINTs activate on any state change, either a transition from HIGH to LOW (falling edge) or from LOW to HIGH (rising edge). Because of this, PCINTs are ideal for monitoring multiple buttons, switches, or sensors without being limited to the dedicated external interrupt pins.

The main components involved in configuring pin-change interrupts are [15]:

- PCICR (Pin Change Interrupt Control Register): Enables the port groups (B, C, or D) that can trigger pin-change interrupts.
- PCMSK0/1/2 (Pin Change Mask Registers): Selects the specific pins within each port that will generate an interrupt.
- ISR(PCINT0/1/2_vect): The interrupt service routines that run when a pin-change event occurs. Inside these ISRs, you must manually determine which pin changed by reading the corresponding PIN register (e.g., PINB, PIND).

The following examples show how to use PCINT, and their sketch codes have been tested with SimulIDE software (Ver1.1.0-SR1, 64-bit, Windows platform).

Example:

This sketch code toggles the onboard LED (pin D13) whenever the button on pin D7 changes state.

```
void setup() {  DDRB |=(1<<PORTB5);        // Set pin D13 (onboard LED) as OUTPUT
```

```
  DDRD &=~(1<<PORTD7);       // Set pin D7 as INPUT
  PORTD |=1<<PORTD7;         // Enable internal pull-up resistor on D7 (button input)
  PCICR |=1<<PCIE2;          // Enable Pin Change Interrupts for port D (PCIE2 bit in PCICR)
  PCMSK2 |=1<<PCINT23;       // Enable pin change interrupt for pin D7 (PCINT23)
  sei();                     // Enable global interrupts
}
void loop() {
  // No code in loop; everything is handled by the interrupt
}
ISR(PCINT2_vect) {
  PORTB ^=(1<<PORTB5);       // Toggle LED on PB5 whenever PD7 changes state
}
/* Sketch uses 508 bytes (1%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.*/
```

Example:

This Arduino sketch code sets up a simple interrupt-based counter using two input buttons connected to digital pins D4 and D5. Inside the interrupt:

- If pin D4 is HIGH (button released, internal pull-up resistor enabled), counter++.
- If pin D5 is HIGH (button released, internal pull-up resistor enabled), counter--.

```
int counter=0;               // Declare a global counter variable initialized to 0
void setup() {
  Serial.begin(9600);        // Initialize serial communication at 9600 baud
  // Instead of pinMode(4, INPUT_PULLUP) and pinMode(5, INPUT_PULLUP):
  // Clear bits 4 and 5 in DDRD to set pins PD4 and PD5 as inputs
  DDRD &= ~( (1<<PD4) | (1<<PD5) );
  // Set bits 4 and 5 in PORTD to enable internal pull-up resistors
  PORTD |= (1<<PORTD4) | (1<<PORTD5);
  PCICR |=B00000100;         // Enable Pin Change Interrupts for port D (PCIE2 bit in PCICR)
  PCMSK2 |=B00110000;        // Enable pin change interrupts for PD4 and PD5 (pins 4 and 5)
  sei();                     // Enable global interrupts
}
void loop() {
  Serial.print("counter :");  // Print label "counter :" to Serial Monitor
  Serial.println(counter);    // Print current counter value
  delay(1000);                // Wait 1000 ms before repeating
}
ISR (PCINT2_vect)            // Interrupt Service Routine for Pin Change Interrupt on port D
{ if(PIND & B00010000)       // If pin PD4 (digital pin 4) is HIGH
    counter=counter + 1;     // Increment counter
  if(PIND & B00100000)       // if pin PD5 (digital pin 5) is HIGH
    counter=counter - 1;     // Decrement counter
}
```

/* Sketch uses 1990 bytes (6%) of program storage space. Maximum is 32256 bytes.
Global variables use 200 bytes (9%) of dynamic memory, leaving 1848 bytes for local variables.
Maximum is 2048 bytes.*/

4.8 Conclusion

This chapter demonstrated how a shift register (74HC595) and a port expander (PCF8574) can significantly extend the ATmega328P's I/O capabilities while reducing pin usage. It also highlighted the importance of interrupts for creating responsive, event-driven systems. Through practical examples, key concepts such as I²C communication, free-running ADC mode, and external interrupt handling were clarified. The next chapter builds on this foundation by introducing the ATmega328P timers used for precise time and signal control.

5.1 Introduction

Timers and counters represent a basic functionality of the ATmega328P microcontroller. This functionality enables a wide range of functions within a sketch. While the Arduino environment relies on them to implement analogWrite() as well as the millis(), micros(), delay(), and delayMicroseconds() functions, it provides less support in terms of direct functionality of the timers for a specific application. Thus, this chapter provides comprehensive insight into the timer and counter modules.

5.2 ATmega328P Timers Overview

The ATmega328P microcontroller has three timers, each with a different size and unique interrupt capabilities. The table below summarizes the timers in the ATmega328P microcontroller [7].

Table 5.1: ATmega328P timers

Timer	Size	Possible Interrupts	Common Uses in Arduino
Timer0	8 bits (0–255)	Compare Match, Overflow	delay(), millis(), micros(), analogWrite() on pins D5, D6.
Timer1	16 bits (0–65,535)	Compare Match, Overflow, Input Capture	Servo functions, analogWrite() on pins D9, D10.
Timer2	8 bits (0–255)	Compare Match, Overflow	tone(), analogWrite() on pins D3, D11.

5.3 Timer Operation with Prescaler in ATmega328P

Below is a diagram of a timer (Timer0) in an ATmega328P microcontroller that increments based on a system clock and a prescaler value (Fig.5.1). In this system, a prescaler is applied to divide a system clock (a 16 MHz system clock, for instance) by a prescaler value of 8, thus ensuring that a timer's slower increment is achieved. Under normal conditions without a prescaler value, a timer will increment for every system clock cycle (at 62.5ns, as per a 16 MHz system clock); in this case, an increment will occur at an interval of 500ns when a prescaler value of 8 is considered. Therefore, this mechanism should be used to generate precise timing intervals, PWM signals, and timer-based interrupts.

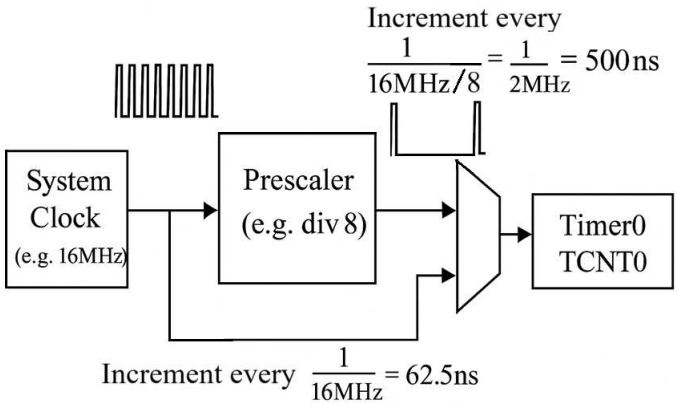


Fig. 5.1: Timer Operation

The following example (Fig.5.2) illustrates the three different operating modes of Timer1 in ATmega328P microcontroller. The 16-bit timer counts upward from 0 to 65535, incrementing with each clock pulse. When the counter value matches a predefined value stored in the Output Compare Register A (for instance, OCR1A = 65531), a compare match interrupt (TIMER1_COMPA) is generated, which can be used for precise timing events. As the timer continues counting, it eventually reaches its maximum value (65535) and rolls over to 0, triggering an overflow interrupt (TIMER1_OVF) to indicate the completion of a full counting cycle. Additionally, when a logic change occurs on the input capture pin (ICP1 on PB0), the current counter value (for example, 3) is stored in the Input Capture Register (ICR1),

generating an input capture interrupt (TIMER1_CAPT) that allows accurate timing of external signals. Together, these events demonstrate how Timer1 can operate in compare match, overflow, and input capture modes for versatile timing and measurement applications [10].

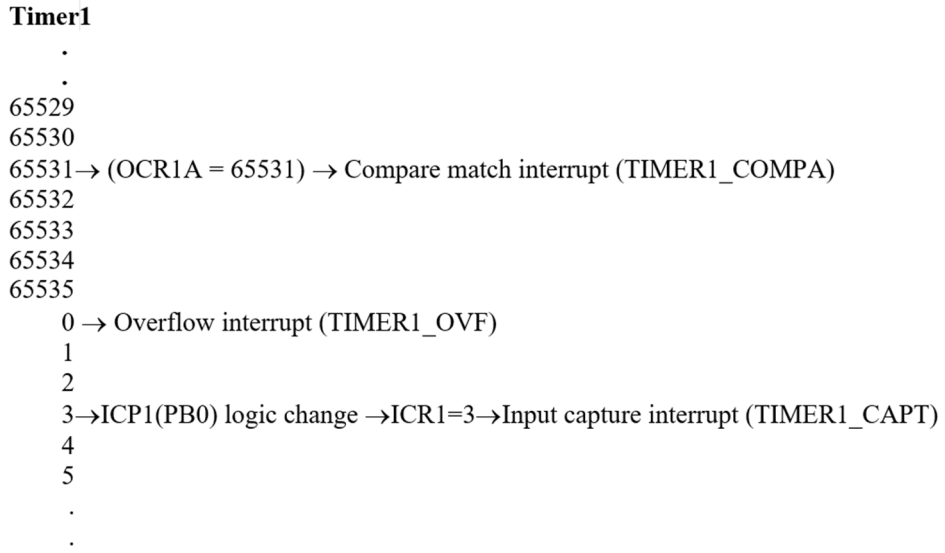


Fig. 5.2: Example of Timer1 Operating Modes

5.4 Timer/Counter1 Registers Description

In this section of the chapter 5, timer1 is selected because it can count from 0 to 65535, offering much higher resolution and a wider timing range compared to the 0–255 range of 8-bit timers (Timer0 and Timer2). Also, timer1 has several dedicated registers that control and monitor its operation and, the following tables describe these registers [24][25].

Table 5.2: Timer1 Main Registers

Register	Description
TCNT1H & TCNT1L	The Timer/Counter Register holds the current 16-bit count value. It's split into high and low bytes: TCNT1 = (TCNT1H << 8) + TCNT1L.
OCR1AH / OCR1AL	The Output Compare Register A holds the 16-bit value that TCNT1 is compared against. When they match, a Compare Match A interrupt can be triggered (TIMER1_COMPA_vect).
OCR1BH / OCR1BL	Same as OCR1A, but for Compare Match B (TIMER1_COMPB_vect), allowing two independent compare operations.
ICR1H / ICR1L	The Input Capture Register stores the timer value (TCNT1) when an external event occurs on the ICP1 pin (PB0). Used in input capture mode for precise event timing.

Table 5.3: Timer1 Control Registers

Register	Description
TCCR1A	Timer/Counter1 Control Register A controls output compare pin behavior (COM1A/B bits) and lower bits of waveform generation mode (WGM10–WGM11).
TCCR1B	Timer/Counter1 Control Register B controls the clock source (CS10–CS12), input capture edge selection (ICES1), and upper bits of waveform generation mode (WGM12–WGM13).
TCCR1C	Timer/Counter1 Control Register C is mainly used to force compare operations (FOC1A/B). Optional in most applications.

Table 5.4: Timer1 Interrupt and Status Registers

Register	Description
TIMSK1	Timer Interrupt Mask Register enables or disables specific Timer1 interrupts (e.g., TOIE1 for overflow, OCIE1A/B for compare match, ICIE1 for input capture).
TIFR1	Timer Interrupt Flag Register holds interrupt flags that indicate whether an interrupt event has occurred. These flags are cleared by writing a logic 1 to them.

5.5 Blinking an LED Using Timer1 Compare Match Mode

The simplest example of toggling the LED on PB5 (D13) without blocking the loop(), so background code can still execute using the Timer1 Compare Match interrupt, is given in this section [15]. Thus, to generate a 500 ms delay using Timer1 on the Arduino Uno (16 MHz clock), we must calculate the value that will be stored in the OCR1A register (Table 5.5). Timer1 increments once every (prescaler / 16 MHz) seconds. So to get 0.5s, the required OCR1A value is:

$$OCR1A = \frac{0.5\,s}{prescaler/16MHz} \qquad (5.1)$$

Table 5.5: Timer1 OCR1A Values for 0.5 s Delay with Different Prescalers

Prescaler	OCR1A calculation	OCR1A value	Fits in 16-bit register?
1	0.5 / (1/16M)	8000000	too large (> 65535)
8	0.5 / (8/16M)	1000000	too large
64	0.5 / (64/16M)	125,000	too large
256	0.5 / (256/16M)	31250	fits perfectly (< 65535)
1024	0.5 / (1024/16M)	7812.5	fits

Therefore, the most suitable prescaler for a 0.5 second compare interval is 256, which gives a perfect integer value (31250) that fits in the 16-bit OCR1A register (Table 5.5).

As shown in Fig.5.3, Timer1 counts clock pulses upward in compare match mode until the counter value equals the value stored in the OCR1A register. When this match occurs, a compare event is generated. In CTC mode (Clear Timer on Compare Match), the timer is automatically reset to zero and an interrupt is triggered. Inside the interrupt service routine, the LED pin is toggled. This mechanism allows the LED to blink with precise timing intervals based on the OCR1A value, without using delay() and without blocking the main program. The output signal on pin PB5, showing the 500 ms toggling period using Timer1 in CTC mode, is illustrated in Fig.5.4.

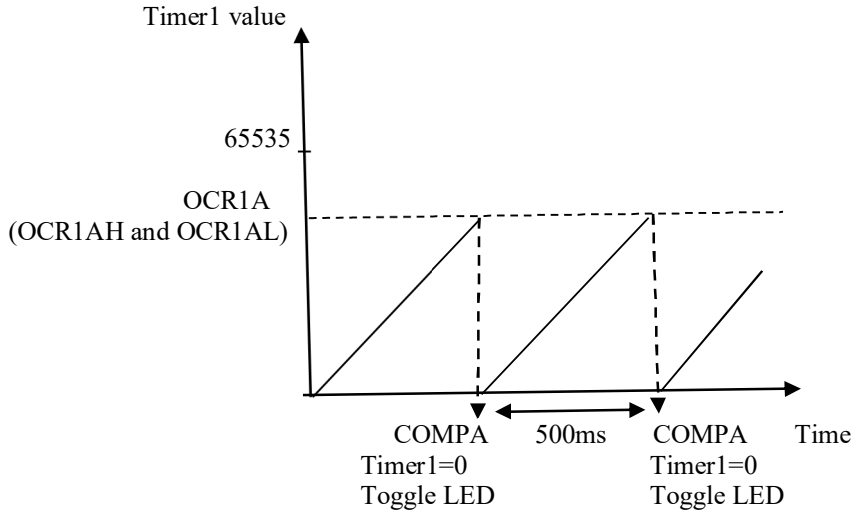


Fig.5.3: Timer1 Compare Match Operation

Example:
This sketch code blinks an LED on PB5 (D13) using Timer1 interrupts for precise timing, without blocking the loop().

```
const int led_pin=PB5;
//counter and compare values
const uint16_t t1_load=0 ;
const uint16_t t1_comp=31250 ;
void setup() {
DDRB |=(1<<led_pin); //Set LED pin to be output
TCCR1A=0x00; //Reset Timer1 control Register A, with WGM11=0, and WGM10=0
//Set CTC mode: Clear Timer on compare. If CTC enabled, then uncomment the following two
//instructions
//TCCR1B |=(1<<WGM12);
//TCCR1B &=~(1<<WGM13);
//Set to prescalar of 256
TCCR1B |=(1<<CS12);
TCCR1B &=~(1<<CS11);
TCCR1B &=~(1<<CS10);
//Reset Timer1 and set compare value
TCNT1=t1_load;
OCR1A= t1_comp ;

//Enable Timer1 compare interrupt
TIMSK1 |=(1<<OCIE1A);
//Enable global interrupt
sei() ;
}
void loop() { //Do nothing !
}
ISR(TIMER1_COMPA_vect)
{
  TCNT1=t1_load;  //if CTC mode is enabled then delete this line.
  PORTB ^=(1<<led_pin) ;
}
/* Sketch uses 556 bytes (1%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables.
Maximum is 2048 bytes.*/
```

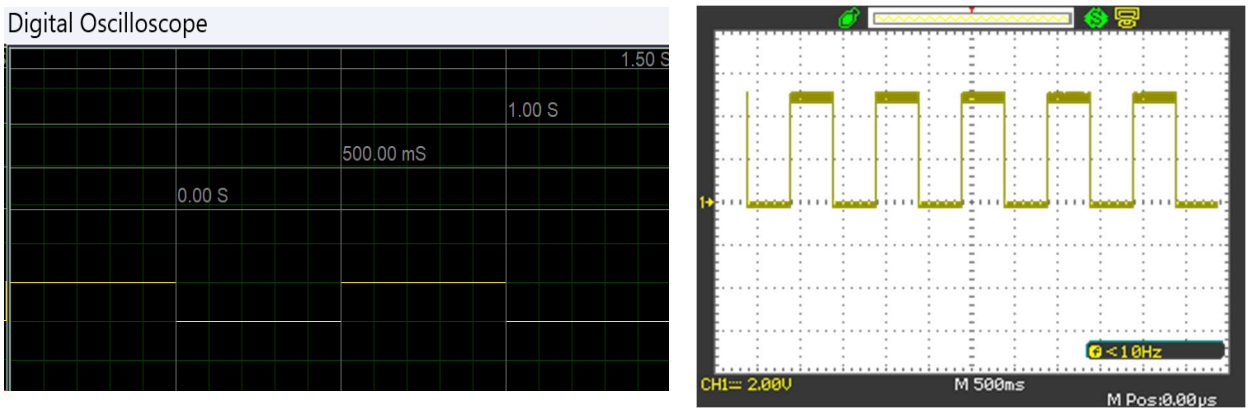


Fig. 5.4: Output signal on pin PB5 (Simulated and Checked with Digital Oscilloscope)

5.6 Timer1 Overflow Mode

Timer1 Overflow is the simplest operating mode of a timer. In this normal mode, the timer counts from the preload value written into TCNT1 up to its maximum. When the counter reaches 65535 and rolls over to 0, an overflow event occurs. If the overflow interrupt is enabled (TOIE1 bit in TIMSK1), the microcontroller automatically calls the ISR(TIMER1_OVF_vect) function [15][24][26]. Inside this ISR we can place any action we want to execute periodically. By choosing a prescaler value and a suitable preload value in TCNT1, we can precisely control how often the overflow happens.

In the following sketch, we use a prescaler of 256 and a preload of 3036 to get exactly 1 second between overflow events. So Timer1 serves as a precise 1 second periodic trigger that toggles pin PD3. The overflow period depends on three parameters: the system clock frequency (Fclock), the prescaler, and the preload value in TCNT1. The preload value can be calculated using the following formula:

$$TCNT1 = 65536 - \frac{Fclock}{prescaler * Ftarget} = 65536 - \frac{16*1000000}{256*1Hz} = 3036 \quad (5.2)$$

Thus, by preloading TCNT1 with 3036, Timer1 overflows exactly once every 1 second. Each overflow triggers the ISR, where the LED on pin D3 is toggled. The measured waveform on pin D3 (Fig.5.5) confirms the operation of this sketch.

```
void setup() {
DDRD |= (1<<PORTD3);    // set PD3 (digital pin 3) as OUTPUT (same as pinMode(3, OUTPUT))
TCCR1A = 0;   // normal operation mode (no PWM)
TCCR1B = 0;   // reset Timer1 control register B
TCNT1 = 3036; // preload counter value: so that Timer1 overflows every 1s with prescaler 256
TCCR1B |= (1<<CS12);   // set prescaler to 256
                              // Timer1 clock = 16MHz / 256 = 62500 Hz →16µs
                             // time = counts * Timer1 clock = 62500*16µs=1s
TIMSK1 |= (1<<TOIE1);   // enable Timer1 overflow interrupt
sei();                          // global interrupt enable
}
// interrupt service routine executed every Timer1 overflow 1000 ms
ISR(TIMER1_OVF_vect)
{
  TCNT1 = 3036;                  // reload counter to get same overflow timing
  PORTD ^= (1<<PORTD3);              // toggle PD3 (pin D3) -> LED ON/OFF
}
void loop() {
  // main loop contains no code, as interrupt handle all required operations.
 }
/* Sketch uses 536 bytes (1%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.*/
```

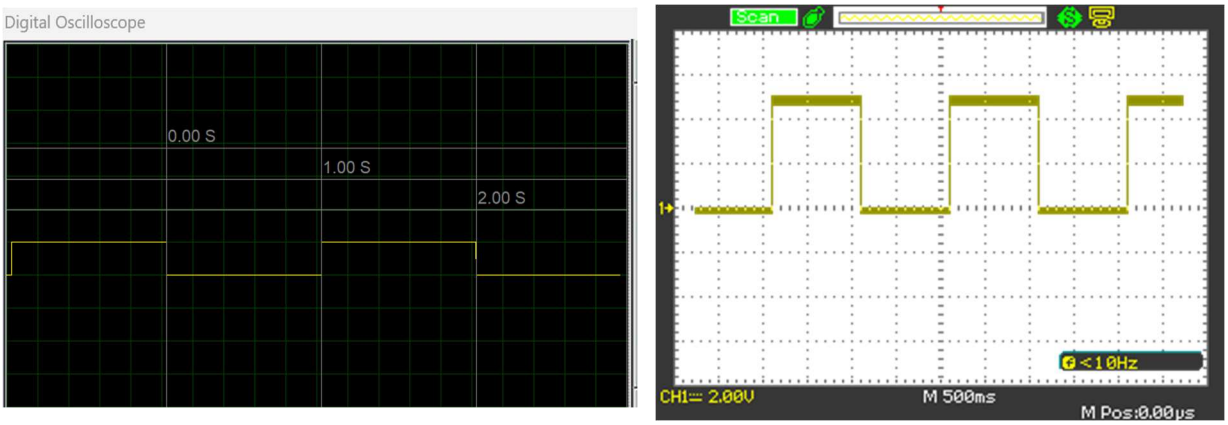


Fig. 5.5: Pin D3 Output Signal (Simulated and Checked with Digital Oscilloscope)

5.7 Timer1 Input Capture Mode

The third Timer1 mode is verified in this section through a simple input-capture example that measures the interval between two falling edges on pin PB0 (ICP1). Each time a falling edge occurs, the hardware copies the timer1 counter value (TCNT1) into the ICR1 register. Inside the ISR, the current capture value is compared with the previous capture value to compute the time difference (delta) between two consecutive falling edges. The result (delta counts and converted milliseconds) is then displayed in the serial monitor at 57600 bauds. A 100 Hz square wave is applied on pin PB0, and the result shown in Fig.5.6 indicates that the measured counts are approximately 156 and the computed time is about 9.98 ms, which confirms the expected value of 10 ms. Also, variables prev and delta hold time measurements updated inside ISR(TIMER1_CAPT_vect), so they must be declared volatile.

```
volatile unsigned int prev = 0;
volatile unsigned int delta = 0;
void setup() {
  // Configure PB0 = input (ICP1 pin)
  DDRB  &= ~(1<<PORTB0);
  Serial.begin(57600);
  cli(); // disable interrupts while configuring
  TCCR1A = 0;
  TCCR1B = 0;
  // ICNC1=1 noise canceler
  // ICES1=0 falling edge
  // CS12:CS10 = 101 prescaler 1024
  TCCR1B = 0b10000101;
  TIMSK1 |= (1<<ICIE1); // enable input capture interrupt
  TCNT1 = 0;          // reset counter
  sei(); // enable interrupts
}
void loop() {
```

```
  delay(1000);
  // convert delta counts to time in ms
  // each tick = 1024/16MHz = 64 µs = 0.064 ms
  float time_ms = delta * 0.064;
  Serial.print("counts = ");
  Serial.print(delta);
  Serial.print("   time = ");
  Serial.print(time_ms);
  Serial.println(" ms");
}
// ISR on falling edge of ICP1 (PB0)
ISR(TIMER1_CAPT_vect)
{
  unsigned int now = ICR1;
  delta = now - prev;   // difference in ticks between this edge and previous edge
  prev = now;
}
/* Sketch uses 3260 bytes (10%) of program storage space. Maximum is 32256 bytes.
Global variables use 228 bytes (11%) of dynamic memory, leaving 1820 bytes for local variables. Maximum is 2048 bytes.*/
```

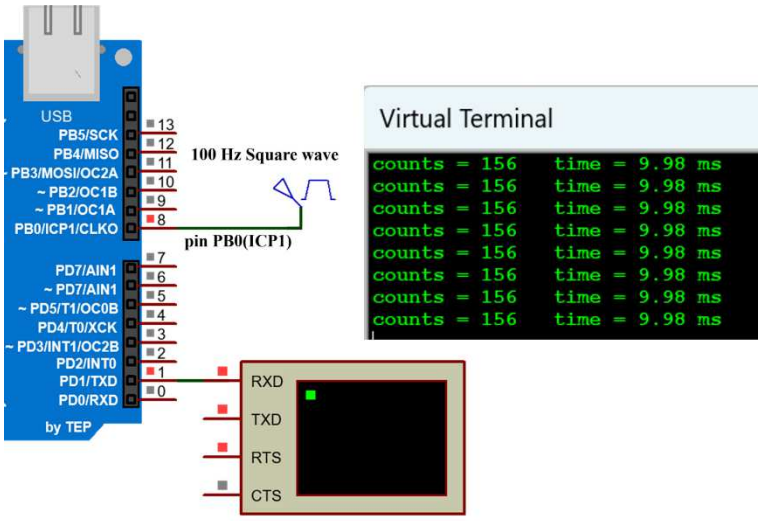


Fig.5.6: Timer1 Input Capture measurement on PB0 (ICP1) using a 100 Hz square wave

5.8 Using Multiple Timers to Generate Signals

The following sketch configures two hardware timers on an Arduino to independently generate square wave signals on digital pins D5 and D6 using interrupts. Timer1 is set in CTC mode with a prescaler of 256 and a compare value of 6250, producing a lower-frequency signal on pin D6 (Fig.5.7), while Timer2 is also configured in CTC mode with a prescaler of 64 and a compare value of 250, generating a higher-frequency signal on pin D5 (Fig.5.8). Each time a timer reaches its compare value, an interrupt triggers an ISR that toggles the corresponding pin, effectively creating two independent periodic output waveforms without using the delay() or loop() functions. As shown in Fig.5.9, this code uses Timer1

(16-bit) to toggle the output on pin D6 every 100 ms, resulting in a frequency of 5 Hz, and Timer2 (8-bit) to toggle the output on pin D5 every 1 ms, resulting in a frequency of approximately 500 Hz.

```
void setup() {
   DDRD|=0b01100000; // DDRD|=(1<<PORTD5)|(1<<PORTD6);
  cli(); // Disable interrupts
  // ---- Timer1 (pin D6) ----
  TCCR1A = 0;
  TCCR1B = 0;
  TCCR1B |= (1 << WGM12); // CTC mode
  TCCR1B |= (1 << CS12);  // Prescaler 256
  OCR1A = 6250;           // Compare value
  TIMSK1 |= (1 << OCIE1A);
  // ---- Timer2 (pin D5) ----
  TCCR2A = 0;
  TCCR2B = 0;
  TCCR2A |= (1 << WGM21); // CTC mode
  TCCR2B |= (1 << CS22); // Prescaler 64
  OCR2A = 250;            // Compare value
  TIMSK2 |= (1 << OCIE2A);
  sei(); // Enable interrupts
}
void loop() {//do nothing!
}
ISR(TIMER1_COMPA_vect) {
  PORTD ^= (1 << PD6); // Toggle pin 6 directly.  Equ. To: led2 = !led2;  digitalWrite(6, led2);
}
ISR(TIMER2_COMPA_vect) {
  PORTD ^= (1 << PD5); //Toggle pin D5 corresponding to: led1 = !led1; digitalWrite(5, led1);
}
/* Sketch uses 620 bytes (1%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes. */
```

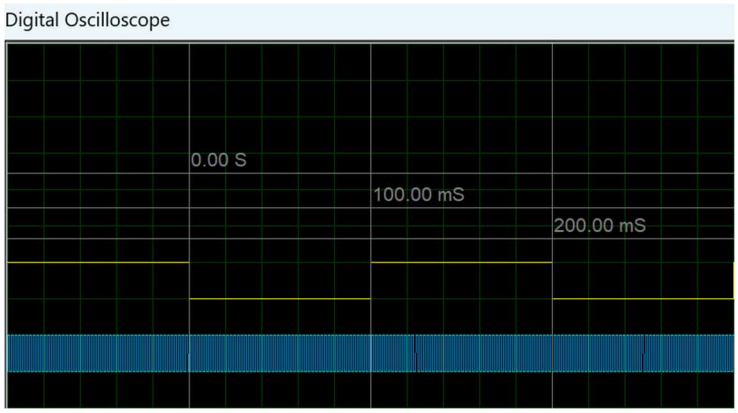


Fig.5.7: Output Signal on Pin D6

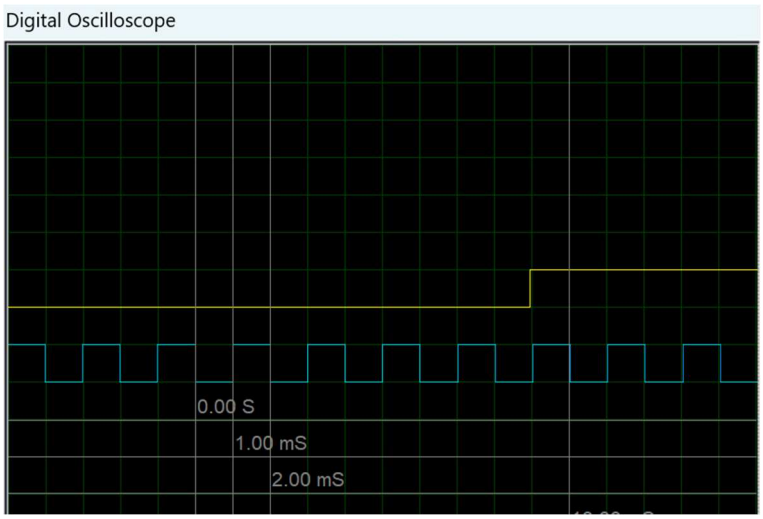


Fig. 5.8: Output Signal on Pin D5

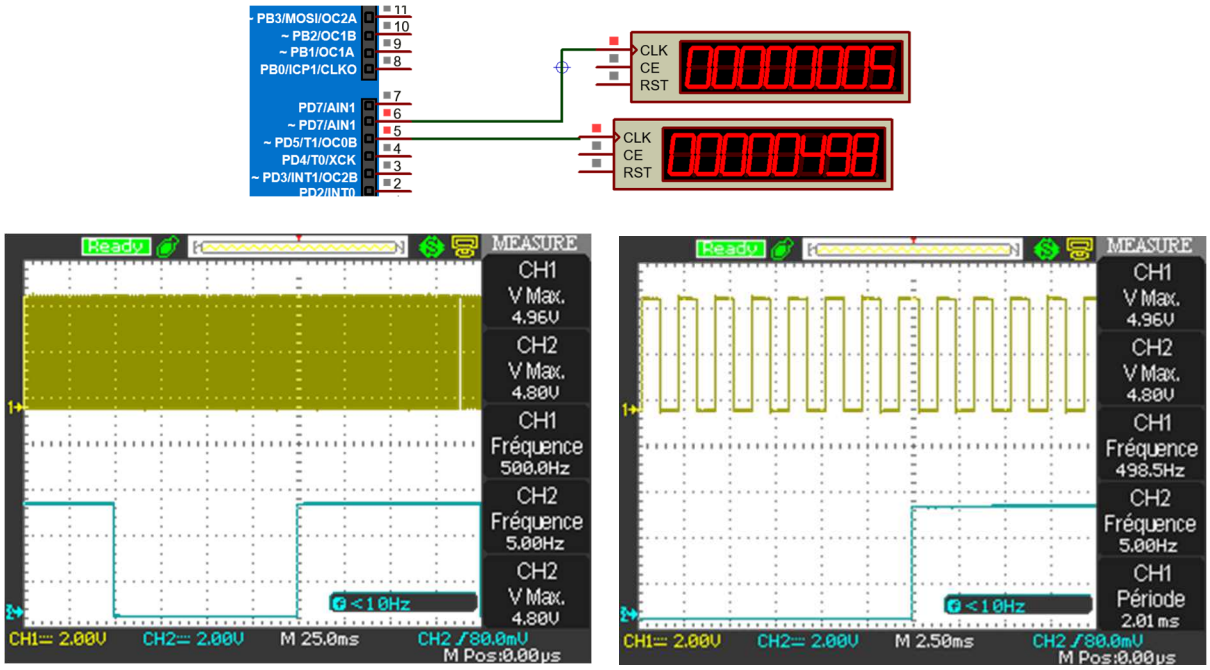


Fig. 5.9: Frequency Measurement of the Output Signals

5.9 Pulse Width Modulation (PWM)

5.9.1 Generating PWM Signals using the analogWrite() Function

PWM is a technique of simulating analog output using digital pulses by rapidly switching a digital pin ON and OFF at a given frequency. It varies the duty cycle to produce average voltages from 0V to 5V. The Uno has PWM on pins 3, 5, 6, 9, 10, and 11 with 8-bit resolution and at frequencies of approximately 490 Hz or 976 Hz (with analogWrite function), depending on which timer is being used.

Register TCCRxB means Timer x, so Timer0 is TCCR0B, Timer1 is TCCR1B, etc.

Syntax: TCCRxB = TCCRxB & 0b11111000 | setting;

where x is the Timer number and setting is the setting used to change the divider.

The following tables (5.6, 5.7, and 5.8) show how the PWM frequency changes based on the prescaler settings for each timer (e.g., Timer0, Timer2) on the ATmega328P [8][26]. All timers use 3 CS bits

(Clock Source: CSx0, CSx1, CSx2) in the TCCRxB register. Each CSxN bit (N = 0–2) controls how the timer gets its clock (prescaler).

Table 5.6: Prescaler settings for Timer0
(TCCR0B register bits: CS00, CS01, CS02, Analog pins: D5, D6)

Setting	Divider	Frequency (HZ)
0x01	1	62500
0x02	8	7812.5
0x03	64	980.5 (default)
0x04	256	244.14
0x05	1024	61.03

Table 5.7: Prescaler settings for Timer1
(TCCR1B register bits: CS10, CS11, CS12, Analog pins: D9, D10)

Setting	Divider	Frequency (HZ)
0x01	1	31372.5
0x02	8	3921.1
0x03	64	490.2 (default)
0x04	256	122.5
0x05	1024	30.6

Table 5.8: Prescaler settings for Timer2
(TCCR2B register bits: CS20, CS21, CS22, Analog pins: D3, D11)

Setting	Divider	Frequency (HZ)
0x01	1	31372.5
0x02	8	3921.1
0x03	32	980.5
0x04	64	490.2 (default)
0x05	128	245.1
0x06	256	122.5
0x07	1024	30.6

Example:
This Arduino sketch generates a PWM signal on pin D9 that smoothly ramps from 0% to 100% duty cycle (0–255) in incremental steps, with a 100 ms delay between each step. After reaching full scale, the PWM output is turned OFF for 2 seconds before repeating the cycle indefinitely. The waveform shown in Fig.5.10 is obtained by setting the PWM frequency to its highest possible value (~31.4 kHz).

```
// Define the PWM output pin connected to the motor
const byte motorPin = 9;
void setup() {
  // Set Timer1 prescaler to 1 (fastest PWM frequency for pin D9 and D10)
  // TCCR1B = Timer/Counter1 Control Register B
  // &0b11111000 clears the lower 3 bits (prescaler bits)
  // |0x01 sets the prescaler to 1 → max frequency ~31.4 kHz using analogWrite function.
```

```
  TCCR1B = TCCR1B & 0b11111000 | 0x01;
}
void loop() {
  // Ramp PWM duty cycle from 0 to 255 in steps of 1
  for (int i = 0; i <= 255; i++) {
    analogWrite(motorPin, i); // Set PWM duty cycle on motorPin
    delay(100);               // Wait 100 ms between steps for smooth ramp
  }
  // Turn OFF the PWM signal
  analogWrite(motorPin, 0);
    // Wait for 2 seconds before starting the next ramp
  delay(2000);
}
/* Sketch uses 1122 bytes (3%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes. */
```

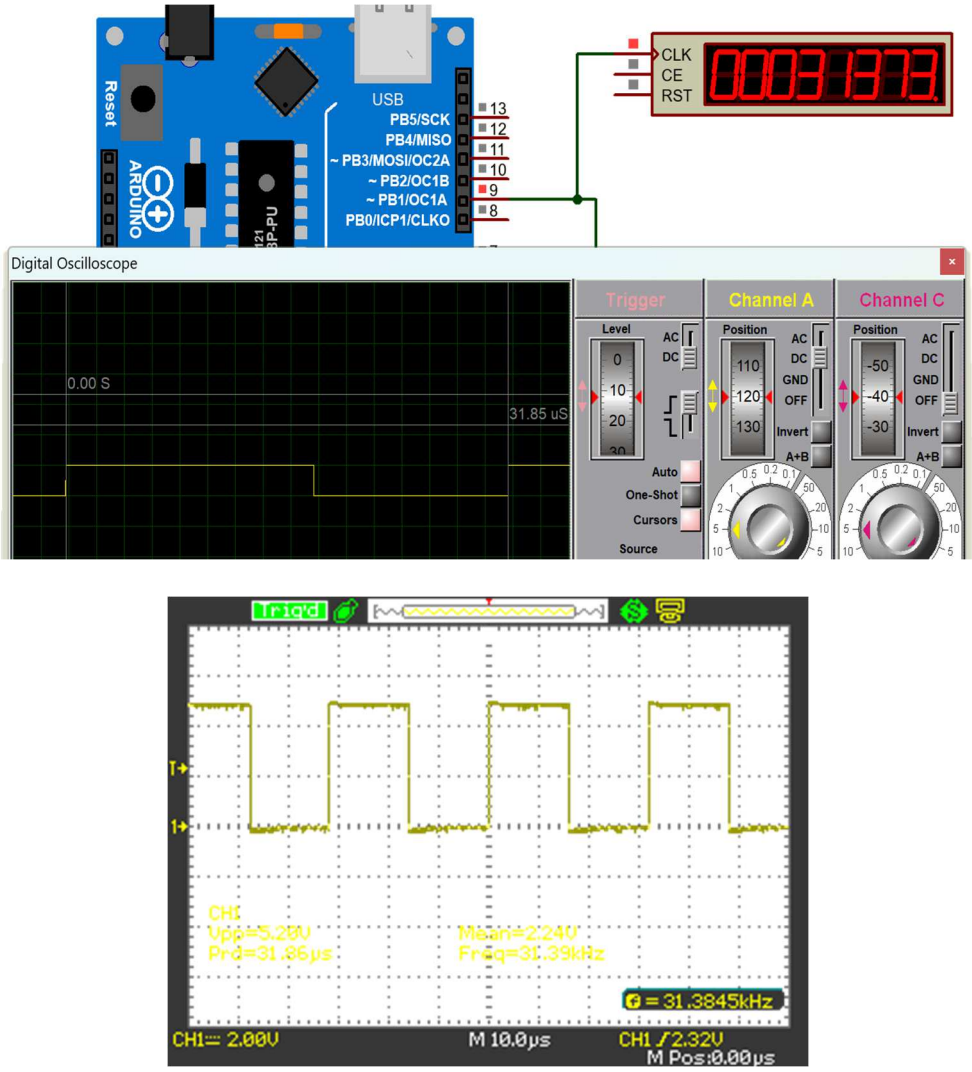


Fig. 5.10: Pin D9 Output Signal Frequency Measurement

5.9.2 Timer1 8-bit Fast PWM Frequency Without Using analogWrite() Function

This Arduino code sets up Timer1 to generate two synchronized 8-bit PWM signals on pins D9 and D10 at ~62.5 kHz (fast PWM, prescaler=1) [8][25][26]. The duty cycle of both PWM outputs is controlled by analog input A0, mapping its 0–1023 range to 0–255. This allows precise, high-speed PWM control using a single potentiometer (Fig.5.11). Both outputs update simultaneously for synchronized operation. Also, the duty cycle ratio is controlled by OCR1A/OCR1B via analogRead() function.

```
void setup() {
  // Set pin D9 (PB1 / OC1A) as output
  DDRB |= (1 << PORTB1);
  // Set pin D10 (PB2 / OC1B) as output
  DDRB |= (1 << PORTB2);
  // Clear Timer/Counter1 control registers A and B to start fresh
  TCCR1A = 0;
  TCCR1B = 0;
  // Configure Timer1:
  // WGM10 = 1, WGM11 = 0  → 8-bit Fast PWM mode (part of mode 5)
  // COM1A1 = 1 → Non-inverting output on OC1A (Pin D9)
  // COM1B1 = 1 → Non-inverting output on OC1B (Pin D10)
  TCCR1A |= (1 << WGM10) | (1 << COM1A1) | (1 << COM1B1);
  // Continue Fast PWM setup:
  // WGM12 = 1, WGM13 = 0 → completes "8-bit Fast PWM" mode
  // CS10 = 1 → No prescaler (Timer1 runs at full CPU clock speed)
  TCCR1B |= (1 << CS10) | (1 << WGM12);
}
void loop() {
  // Read analog input from A0 (10-bit value: 0–1023)
  // Map it to an 8-bit PWM value: 0–255
  int pwmValue = map(analogRead(A0), 0, 1023, 0, 255);
  // Update duty cycle for both PWM channels:
  // OCR1A → controls OC1A (D9)
  // OCR1B → controls OC1B (D10)
  OCR1A = pwmValue;
  OCR1B = pwmValue;
delay(200);
}
/*Sketch uses 896 bytes (2%) of program storage space. Maximum is 32256 bytes.
Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables.
Maximum is 2048 bytes. */
```

For 8-bit fast PWM, frequency is given by:

$$f_{PWM} = \frac{f_{CPU}}{N \times 256} = 62.5\text{KHz} \qquad (5.3)$$

where:

- f_{CPU}=16 MHz
- prescaler N= 1

- 256 = TOP value in 8-bit PWM mode

The following table shows how different Timer1 prescaler settings on a 16 MHz Arduino produce PWM frequencies from 62.5 kHz down to 61 Hz in 8-bit Fast PWM mode.

Table 5.9: Timer1 8-bit Fast PWM Frequencies vs. Prescaler

CS12	CS11	CS10	Prescaler (N)	PWM Frequency
0	0	1	1	62.5 kHz
0	1	0	8	7.8125 kHz
0	1	1	64	976.6 Hz
1	0	0	256	244.1 Hz
1	0	1	1024	61.0 Hz

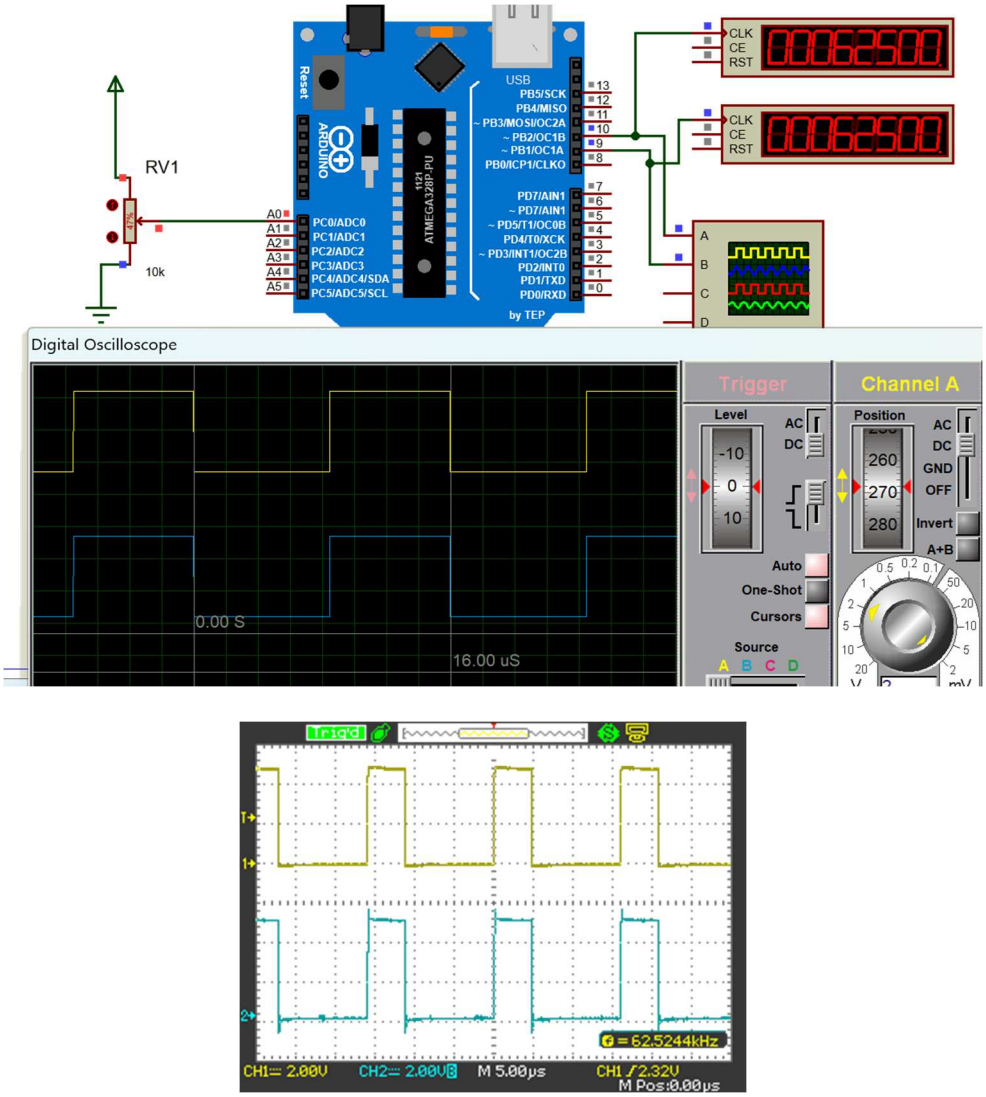


Fig. 5.11: Generation of PWM signals on pins D9 and D10 using Timer1

5.9.3 10-bit Fast PWM Using Timer1

In 10-bit Fast PWM, Timer1 counts from 0 to1023 (instead of 255), giving smoother duty-cycle resolution but lower frequency [24][25]. Thus, the corresponding frequency table and Arduino sketch code for pin D9 (OC1A) (and optionally D10 / OC1B) are given below.

Table 5.10: Timer1 10-bit Fast PWM Frequencies vs. Prescaler

CS12	CS11	CS10	Prescaler (N)	PWM Frequency
0	0	1	1	15.625 kHz
0	1	0	8	1.953 kHz
0	1	1	64	244.14 Hz
1	0	0	256	61.04 Hz
1	0	1	1024	15.26 Hz

Thus, the the PWM frequency formula is as follow:

$$f_{PWM} = \frac{f_{CPU}}{N \times 1024} \quad (5.4)$$

All frequency values shown by table 5.10 with their CSxx bits, have been checked with an Arduino Uno board and an oscilloscope. Thus, the corresponding 10-bit PWM signals are given in Fig.5.12.

```
void setup() {
  // Set pins D9 (PB1 / OC1A) and D10 (PB2 / OC1B) as outputs
  DDRB |= (1 << PORTB1) | (1 << PORTB2);
  // Clear Timer1 control registers
  TCCR1A = 0;
  TCCR1B = 0;
  // Configure Timer1 for 10-bit Fast PWM:
  // WGM10=1, WGM11=1, WGM12=1, WGM13=0 → Fast PWM, 10-bit (Mode 7)
  // COM1A1=1 → Non-inverting output on OC1A (D9)
  // COM1B1=1 → Non-inverting output on OC1B (D10)
  TCCR1A |= (1 << WGM10) | (1 << WGM11) | (1 << COM1A1) | (1 << COM1B1);
  TCCR1B |= (1 << WGM12);
  // Set prescaler → 1 (no prescaling) for 15.625 kHz PWM
  TCCR1B |= (1 << CS10);
  Serial.begin(115200);
}
void loop() {
  // Read analog input (0–1023) and map directly to 10-bit PWM (0–1023)
  //int pwmValue = analogRead(A0);
  for(int i=0;i<=1023;i++)
  {
  // Apply duty cycle to both PWM channels
  OCR1A = i;  // D9
  OCR1B = 1023-i;  // D10
  Serial.println(i);
  delay(200);
  }
}/*Sketch uses 1884 bytes (5%) of program storage space. Maximum is 32256 bytes. Global variables use 188 bytes (9%) of dynamic memory, leaving 1860 bytes for local variables. Maximum is 2048 bytes.*/
```

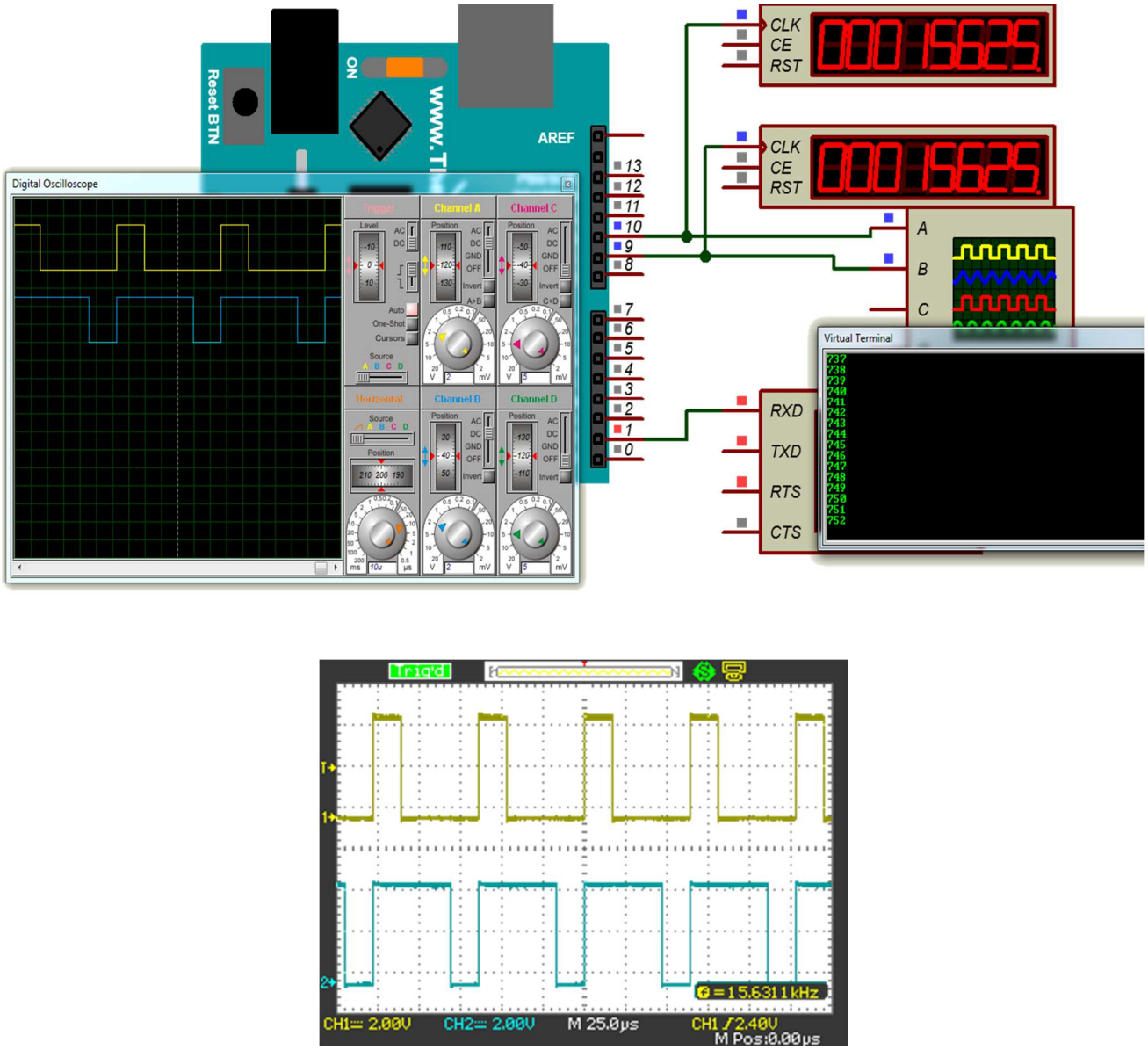


Fig. 5.12: Generation of 10-bit PWM Signals on Pins D9 and D10 using Timer1

5.10 Conclusion

Timers in the ATmega328P microcontroller are important tools that help manage time-related tasks like delays, counting events, creating signals, and controlling PWM. Understanding their operation modes such as compare match, overflow, and input capture, enables efficient implementation of both simple and advanced tasks without excessive CPU intervention. By mastering timer configurations and prescaler settings, developers can create reliable timing functions, generate accurate waveforms, and optimize overall system performance in embedded applications.

Conclusion

This manual has delivered a rigorous, technically oriented examination of the ATmega328P microcontroller within the Arduino development environment, progressing systematically from foundational principles to practical system-level implementation. Through in-depth coverage of programming architecture, digital and analog I/O subsystems, timing and clock management, interrupt handling, peripheral communication protocols, and sensor data acquisition methodologies, the reader acquires a solid and operational understanding of embedded system design.

By integrating theory with hands-on examples and project-oriented tasks, this manual equips students with the analytical skills and technical competencies necessary to design, implement, and troubleshoot microcontroller-based systems. The knowledge acquired serves as a solid foundation for more advanced studies in embedded systems, control engineering, digital communication, and IoT development. Ultimately, the aim of this manual is to inspire further exploration, innovation, and confidence in creating reliable and efficient embedded solutions using the ATmega328P and the broader Arduino platform.

Finally, it is important to note that displays such as OLED and TFT modules, with or without touch screens, as well as digital sensors that rely on more complex communication protocols, are outside the scope of this document. These components are excluded because they require more advanced protocol handling compared with analog sensors. Additionally, the reason for using an older version of the Arduino IDE is that all releases of the legacy Arduino IDE 1.8.x support both 32-bit and 64-bit Windows systems, including Windows 7 and newer, and the computing infrastructure available within our department is still based on 32-bit systems.

References

[1] Arduino, Arduino® UNO R3 Product Reference Manual, SKU: A000066.
https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf
[2] Arduino, Arduino Uno Rev. 3 schematic, Arduino-CC.
https://www.arduino.cc/en/uploads/Main/Arduino_Uno_Rev3-schematic.pdf
[3] Arduino, Arduino UNO R3 Full Pinout, SKU: A000066.
https://docs.arduino.cc/resources/pinouts/A000066-full-pinout.pdf
[4] Arduino, Arduino Nano Product Reference Manual, SKU: A000005.
https://docs.arduino.cc/resources/datasheets/A000005-datasheet.pdf
[5] Arduino, "Arduino Hardware."
https://www.arduino.cc/en/hardware
[6] S. Hassan, Arduino Masterclass Build Electronics Projects from Scratch, Independently Published, 2025.
[7] Microchip Technology Inc., AVR® Microcontroller with picoPower® Technology ATmega328/P, Datasheet, Doc. No. DS40001984A, 2018.
[8] C. M. Amariei, Arduino Development Cookbook: Over 50 Hands-on Recipes to Quickly Build and Understand Arduino Projects. Birmingham, UK: Packt Publishing, 2015. doi: 10.1007/978-1-78398-2950-4
[9] Arduino, "Arduino Documentation: Language Reference."
https://docs.arduino.cc/language-reference/
uage-reference/en/functions/interrupts/interrupts/
[10] S. F. Barrett, Arduino II: Systems, Springer Cham, Switzerland, 2020.
doi: 10.1007/978-3-031-79919-8
[11] B. Evans, Beginning Arduino Programming. Berkeley, CA: Apress, 2011.
doi: 10.1007/978-1-4302-3778-5
[12] Arduino, "Arduino Documentation: Arduino as ISP and Arduino Bootloaders."
https://docs.arduino.cc/built-in-examples/arduino-isp/ArduinoISP/
[13] Arduino, "Arduino Documentation: From Arduino to a Microcontroller on a Breadboard."
https://docs.arduino.cc/built-in-examples/arduino-isp/ArduinoToBreadboard/
[14] S. Monk, Programming Arduino: Getting Started with Sketches, 3rd ed. New York, NY, USA: McGraw-Hill Education, 2023.
[15] N. Dunbar, Arduino Interrupts: Harness the Power of Interrupts in Your Arduino and ATmega328 Code, Berkeley, CA: Apress, 2024.
doi: 10.1007/978-1-4842-9714-8
[16] M. Geddes, Arduino Project Handbook: 25 Practical Projects to Get You Started. San Francisco, CA: No Starch Press, 2016.
[17] Arduino, "Arduino Documentation: Liquid Crystal Displays (LCD) with Arduino."
https://docs.arduino.cc/learn/electronics/lcd-displays/
[18] D. Ibrahim, "Microcontroller-Based Temperature Monitoring and Control", 1st ed. London, U.K.: Newnes, 2002.
doi.org/10.1016/B978-0-7506-5556-9.X5000-2
[19] P. Furquim, "Temperature Sensor with Arduino and NTC 10k Sensor," GitHub repository, 2023.
https://github.com/devpedrofurquim/temperature-sensor-with-arduino
[20] S. Monk, "Programming Arduino Next Steps: Going Further with Sketches", 2nd ed. New York, NY, USA: McGraw-Hill Education, 2019.
[21] D. Senne, "Simple Kalman Filter," GitHub repository, 2024.

https://github.com/denyssene/SimpleKalmanFilter.
[22] NXP Semiconductors, “PCF8574; PCF8574A: Remote 8-bit I/O expander for I^2C-bus with interrupt”, 2013.
https://www.nxp.com/docs/en/data-sheet/PCF8574_PCF8574A.pdf
[23] J. Blum, Exploring Arduino: Tools & Techniques for Engineering Wizardry, 2nd Edition. Indianapolis, USA: Wiley, 2019.
doi: 10.1002/9781119405320
[24] S. F. Barrett, “Arduino Microcontroller: Processing for Everyone!”, 3rd ed., Morgan & Claypool Publishers, 2013.
doi: 10.2200/S00522ED1V01Y201307DCS043
[25] D. J. Russell, “Introduction to Embedded Systems: Using ANSI C and the Arduino Development Environment”, Morgan & Claypool Publishers, 2010.
[26] M. Margolis, Arduino Cookbook, 2nd Edition. Sebastopol, CA: O’Reilly Media, 2012.

Annex 1

The interrupt vectors of the ATmega328P are stored in flash memory starting at address 0x0000. Each vector corresponds to its interrupt service routine (ISR). When an interrupt occurs, the CPU retrieves the appropriate vector from flash and branches to the associated ISR, which is written by the developer to handle the specific event.

Table A.1: ATmega328P Reset and Interrupt Vectors

Vector Nº	Address	Source	ISR Name	Interrupts Definition
1	0x0000	RESET	RESET_vect	External Pin, Power-on Reset, Brown-out Reset, Watchdog System Reset
2	0x0002	INT0	INT0_vect	External Interrupt Request 0
3	0x0004	INT1	INT1_vect	External Interrupt Request 1
4	0x0006	PCINT0	PCINT0_vect	Pin Change Interrupt Request 0
5	0x0008	PCINT1	PCINT1_vect	Pin Change Interrupt Request 1
6	0x000A	PCINT2	PCINT2_vect	Pin Change Interrupt Request 2
7	0x000C	WDT	WDT_vect	Watchdog Timeout Interrupt
8	0x000E	TIMER2_COMPA	TIMER2_COMPA_vect	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	TIMER2_COMPB_vect	Timer/Counter2 Compare Match B
10	0x0012	TIMER2_OVF	TIMER2_OVF_vect	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	TIMER1_CAPT_vect	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
14	0x001A	TIMER1_OVF	TIMER1_OVF_vect	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	TIMER0_COMPA_vect	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	TIMER0_COMPB_vect	Timer/Counter0 Compare Match B
17	0x0020	TIMER0_OVF	TIMER0_OVF_vect	Timer/Counter0 Overflow
18	0x0022	SPI_STC	SPI_STC_vect	SPI Serial Transfer Complete
19	0x0024	USART_RX	USART_RX_vect	USART Rx Complete
20	0x0026	USART_UDRE	USART_UDRE_vect	USART Data Register Empty
21	0x0028	USART_TX	USART_TX_vect	USART Tx Complete
22	0x002A	ADC	ADC_vect	ADC Conversion Complete
23	0x002C	EEPROM_READY	EE_READY_vect	EEPROM Ready
24	0x002E	ANALOG_COMP	ANALOG_COMP_vect	Analog Comparator
25	0x0030	TWI	TWI_vect	2-Wire Serial Interface (I^2C)
26	0x0032	SPM_READY	SPM_READY_vect	Store Program Memory Ready

Annex 2

This program configures the ATmega328P at the register level to handle I²C LCD control, ADC temperature measurement using the 1.1 V reference, UART transmission at 57600 bps, and a custom Timer0-based millis() function. It repeatedly samples the temperature every second, displays it on the LCD, and sends the value over UART.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <stdlib.h>
//================================
// LCD (I²C via PCF8574) constants
//================================
#define LCD_ADDR 0x27          // I²C address of LCD module
#define I2C_WRITE 0
#define LCD_BACKLIGHT 0x08
#define ENABLE 0x04
#define RS 0x01
//================================
// Global variable for millis counter
//================================
volatile unsigned long millis_count = 0;
//================================
// Function Prototypes
//================================
void Timer0_Init(void);
unsigned long millis_custom(void);
void I2C_Init(void);
void I2C_Start(void);
void I2C_Stop(void);
void I2C_Write(uint8_t data);
void LCD_SendNibble(uint8_t data);
void LCD_Cmd(uint8_t cmd);
void LCD_Char(char data);
void LCD_Init(void);
void LCD_String(const char *str);
void LCD_SetCursor(uint8_t col, uint8_t row);
void ADC_Init(void);
uint16_t ADC_Read(uint8_t channel);
void UART_Init(void);
void UART_SendChar(char c);
void UART_SendString(const char *s);
void UART_SendFloat(float val);
//================================
// Setup (runs once)
```

```
//================================
void setup() {
  cli();            // Disable global interrupts during setup
    DDRB |= (1 << PB5);   // Make PB5 (D13) an output
  PORTB &= ~(1 << PB5); // Drive it LOW (LED off)
    UART_Init();       // Initialize UART for serial communication
  I2C_Init();        // Initialize I²C (TWI)
  LCD_Init();        // Initialize LCD via I²C
  ADC_Init();        // Initialize ADC with internal 1.1V reference
  Timer0_Init();     // Initialize Timer0 for 1ms interrupt
  sei();             // Enable global interrupts
}
//================================
// Loop (runs forever)
//================================
void loop() {
  static unsigned long prev_time = 0;
  const unsigned long period = 1000; // Sample every 1000 ms (1 second)
  unsigned long now = millis_custom(); // Our own millis() function
  if ((now - prev_time) >= period) {
    prev_time = now;
    uint32_t data = 0;
    for (uint8_t i = 0; i < 100; i++) {
      data += ADC_Read(0); // Read from ADC0 (A0)
    }
    // Convert ADC to temperature (assuming 1.1V internal reference)
    float temp = (float)data * 1.1 / 1024.0;
    // --- UART Output ---
    UART_SendFloat(temp);
    UART_SendString("°C\r\n");
    // --- LCD Output ---
    LCD_SetCursor(3, 0);
    LCD_String("Temperature");
    LCD_SetCursor(5, 1);
    char buf[10];
    dtostrf(temp, 4, 1, buf);
    LCD_String(buf);
    LCD_Char(0xDF); // Degree symbol
    LCD_Char('C');
  }
}
//================================
// Timer0 Initialization (1 ms tick)
//================================
void Timer0_Init(void) {
  // CTC mode: WGM01 = 1, WGM00 = 0
```

```
  TCCR0A = (1 << WGM01);
  // Prescaler = 64 → CS01 and CS00 bits set
  TCCR0B = (1 << CS01) | (1 << CS00);
  // Compare value for 1ms tick (16MHz / 64 / 1000 - 1 = 249)
  OCR0A = 249;
  // Enable Output Compare Match A Interrupt
  TIMSK0 = (1 << OCIE0A);
}
// Interrupt Service Routine — runs every 1 ms
ISR(TIMER0_COMPA_vect) {
  millis_count++; // Increment our custom milliseconds counter
}
// Return the current milliseconds count
unsigned long millis_custom(void) {
  unsigned long temp;
  cli();
  temp = millis_count;
  sei();
  return temp;
}
//==============================
// UART (Serial) Functions
//==============================
void UART_Init(void) {
  uint16_t ubrr = (F_CPU / 16 / 57600) - 1;  // Baud rate calculation
  UBRR0H = (uint8_t)(ubrr >> 8);
  UBRR0L = (uint8_t)ubrr;
  UCSR0B = (1 << TXEN0);                    // Enable transmitter
  UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);       // 8 data bits, no parity, 1 stop
}
void UART_SendChar(char c) {
  while (!(UCSR0A & (1 << UDRE0)));  // Wait for empty transmit buffer
  UDR0 = c;                         // Send character
}
void UART_SendString(const char *s) {
  while (*s) UART_SendChar(*s++);
}
void UART_SendFloat(float val) {
  char buffer[10];
  dtostrf(val, 4, 2, buffer);
  UART_SendString(buffer);
}
//==============================
// I²C (TWI) Functions
//==============================
void I2C_Init(void) {
```

```
  TWSR = 0x00;  // Prescaler = 1
  TWBR = ((F_CPU / 100000UL) - 16) / 2;  // 100kHz SCL
}
void I2C_Start(void) {
  TWCR = (1 << TWSTA) | (1 << TWEN) | (1 << TWINT);
  while (!(TWCR & (1 << TWINT)));  // Wait for start condition complete
}
void I2C_Stop(void) {
  TWCR = (1 << TWSTO) | (1 << TWINT) | (1 << TWEN);
  while (TWCR & (1 << TWSTO));     // Wait for stop condition
}
void I2C_Write(uint8_t data) {
  TWDR = data;
  TWCR = (1 << TWEN) | (1 << TWINT);
  while (!(TWCR & (1 << TWINT)));  // Wait for data transmitted
}
//================================
// LCD over I²C (4-bit mode)
//================================
void LCD_SendNibble(uint8_t data) {
  I2C_Start();
  I2C_Write((LCD_ADDR << 1) | I2C_WRITE);
  I2C_Write(data | ENABLE | LCD_BACKLIGHT);
  _delay_us(1);
  I2C_Write((data & ~ENABLE) | LCD_BACKLIGHT);
  I2C_Stop();
}
void LCD_Cmd(uint8_t cmd) {
  LCD_SendNibble(cmd & 0xF0);
  LCD_SendNibble((cmd << 4) & 0xF0);
  _delay_ms(2);
}
void LCD_Char(char data) {
  uint8_t high = (data & 0xF0) | RS;
  uint8_t low = ((data << 4) & 0xF0) | RS;
  LCD_SendNibble(high);
  LCD_SendNibble(low);
}
void LCD_Init(void) {
  _delay_ms(50);
  LCD_Cmd(0x33);
  LCD_Cmd(0x32);
  LCD_Cmd(0x28); // 4-bit, 2-line, 5x8 font
  LCD_Cmd(0x0C); // Display ON, cursor OFF
  LCD_Cmd(0x06); // Entry mode, increment cursor
  LCD_Cmd(0x01); // Clear display
```

```
  _delay_ms(5);
}
void LCD_String(const char *str) {
  while (*str) LCD_Char(*str++);
}
void LCD_SetCursor(uint8_t col, uint8_t row) {
  uint8_t row_offsets[] = {0x00, 0x40, 0x14, 0x54};
  LCD_Cmd(0x80 | (col + row_offsets[row]));
}
//================================
// ADC (Analog-to-Digital Converter)
//================================
void ADC_Init(void) {
  // Internal 1.1V reference → REFS1=1, REFS0=1
  ADMUX = (1 << REFS1) | (1 << REFS0);
  // Enable ADC, prescaler = 128 (ADPS2:0 = 111)
  ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
}
uint16_t ADC_Read(uint8_t channel) {
  ADMUX = (ADMUX & 0xF0) | (channel & 0x0F);  // Select ADC channel
  ADCSRA |= (1 << ADSC);                      // Start conversion
  while (ADCSRA & (1 << ADSC));               // Wait until complete
  return ADC;                                 // Return 10-bit result
}
/* Sketch uses 3232 bytes (10%) of program storage space. Maximum is 32256 bytes.
Global variables use 35 bytes (1%) of dynamic memory, leaving 2013 bytes for local variables. Maximum is 2048 bytes. */
```

Annex 3

This Arduino sketch implements a simple yet robust temperature control system using an NTC thermistor for precise temperature sensing via the Steinhart-Hart equation, with 200-sample averaging to reduce noise (Fig.A31). It features a 128x64 OLED display showing real-time temperature, setpoint, and error values, while RGB LEDs provide intuitive visual feedback (green = on target, red = too cold, blue = too hot) (Fig.A32). Two interrupt-driven buttons allow users to adjust the temperature setpoint between 0°C and 40°C in 0.5°C steps. The core control logic uses a classic ON/OFF algorithm with a 200ms update cycle to drive a heating element via a digital output pin, and all data is streamed over serial for external monitoring or logging.

```
// Include the Wire library for I²C communication (used by OLED display)
#include <Wire.h>
// Include the Adafruit GFX library for graphics primitives and text rendering
#include <Adafruit_GFX.h>
// Include the Adafruit SSD1306 library for controlling the OLED display module
#include <Adafruit_SSD1306.h>

// ======================== Pin Definitions ===================
// Define digital pin 9 as the output pin for controlling the heating/cooling device
#define CONTROL_OUTPUT_PIN 9
// Define digital pin 5 as the output pin for the green LED indicator
#define LED_GREEN_PIN      5
// Define digital pin 4 as the output pin for the red LED indicator
#define LED_RED_PIN        4
// Define digital pin 6 as the output pin for the blue LED indicator
#define LED_BLUE_PIN       6
// Define digital pin 3 as the input pin for the UP button (with internal pull-up)
#define BUTTON_UP_PIN      3
// Define digital pin 2 as the input pin for the DOWN button (with internal pull-up)
#define BUTTON_DOWN_PIN    2

// ======================== OLED SETTINGS ====================
// Define the width of the OLED display in pixels (standard 128x64 SSD1306)
// OLED→Arduino Uno: VCC→3.3V, GND→GND, SCL→A5, SDA→A4
#define SCREEN_WIDTH  128
// Define the height of the OLED display in pixels (standard 128x64 SSD1306)
#define SCREEN_HEIGHT 64
// Create an Adafruit_SSD1306 display object using I2C (Wire) and no reset pin (-1)
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

// Global variable to store the current measured temperature in Celsius
float temperatureC = 0.0f;
// Global variable to store the current control error (setpoint - measured temperature)
float error = 0.0f;
// Volatile global variable for the temperature setpoint (modified in ISR, so volatile)
volatile float setpoint = 24.0f;
```

```
// ========================= NTC Model constants  ========================
// Define the fixed resistor value (10kΩ) in the voltage divider with the NTC thermistor
const float R1 = 10000.0f;
// Define Steinhart-Hart coefficient c1 for NTC temperature conversion
const float c1 = 0.001129148f,
// Define Steinhart-Hart coefficient c2 for NTC temperature conversion
        c2 = 0.000234125f,
// Define Steinhart-Hart coefficient c3 for NTC temperature conversion
        c3 = 0.0000000876741f;

// ========================= Timing & Precomputed Constants ====================
// Define the control loop period in milliseconds (200ms = 5Hz update rate)
const unsigned long CONTROL_PERIOD_MS = 200;
// Define the display update period in milliseconds (200ms = 5Hz refresh rate)
const unsigned long DISPLAY_PERIOD_MS = 200;
// Global variable to store the last time the control loop executed (for non-blocking timing)
unsigned long last_control_time = 0;
// Global variable to store the last time the display was updated (for non-blocking timing)
unsigned long last_display_time = 0;

// Global boolean variable to track the current state of the control output (ON/OFF)
bool controlOutputBool = false;

// ========================= Function Forward Declarations ====================
// Apply ON/OFF control logic based on temperature vs setpoint
void applyOnOff();
// Update LED indicators based on current error value
void updateLEDs();
// Update the OLED display with current temperature, setpoint, and error
void updateDisplay();
// Measure temperature using NTC thermistor and Steinhart-Hart equation
float measur_temp();

// ========================= SETUP ====================
// Setup function: runs once at startup to initialize hardware and peripherals
void setup() {
  // Configure CONTROL_OUTPUT_PIN as an output pin for driving the heater
  pinMode(CONTROL_OUTPUT_PIN, OUTPUT);
  // Initialize CONTROL_OUTPUT_PIN to LOW state (turn off heating element initially)
  digitalWrite(CONTROL_OUTPUT_PIN, LOW);
  // Configure LED_GREEN_PIN as an output pin for status indication
  pinMode(LED_GREEN_PIN, OUTPUT);
  // Configure LED_RED_PIN as an output pin for status indication
  pinMode(LED_RED_PIN,   OUTPUT);
  // Configure LED_BLUE_PIN as an output pin for status indication
```

```
pinMode(LED_BLUE_PIN,  OUTPUT);
// Configure BUTTON_UP_PIN as input with internal pull-up resistor enabled
pinMode(BUTTON_UP_PIN,   INPUT_PULLUP);
// Configure BUTTON_DOWN_PIN as input with internal pull-up resistor enabled
pinMode(BUTTON_DOWN_PIN, INPUT_PULLUP);

//=============== Interrupts on Pins D2 and D3=========
//Attach interrupt to BUTTON_DOWN_PIN (D2) to trigger Interrupt_BUTTON_DOWN on
//FALLING edge (button press)
attachInterrupt(digitalPinToInterrupt(BUTTON_DOWN_PIN), Interrupt_BUTTON_DOWN, FALLING);

// Attach interrupt to BUTTON_UP_PIN (D3) to trigger Interrupt_BUTTON_UP on FALLING edge
//(button press)
attachInterrupt(digitalPinToInterrupt(BUTTON_UP_PIN), Interrupt_BUTTON_UP, FALLING);

// Initialize serial communication at 115200 baud rate for debugging and data logging
Serial.begin(115200);

// Initialize the OLED display with SSD1306_SWITCHCAPVCC voltage and I2C address 0x3C
// If initialization fails, enter an infinite loop to halt execution (error handling)
if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
  for (;;); // Halt program if display initialization fails
}
// Short delay to allow display to stabilize after initialization
delay(100);

// Clear the display buffer to remove any residual content
display.clearDisplay();
// Set the text color to white (SSD1306_WHITE) for all subsequent text rendering
display.setTextColor(SSD1306_WHITE);
// Set the text size to 1x (normal size) for the initialization message
display.setTextSize(1);
// Position the cursor at column 0, row 30 (pixel coordinates) for centered text
display.setCursor(0, 30);
// Print the initialization message to the display buffer
display.print("Initializing...");
// Send the buffered content to the physical OLED display to make it visible
display.display();

  // Turn OFF the green LED initially
digitalWrite(LED_GREEN_PIN, LOW);
// Turn OFF the red LED initially
digitalWrite(LED_RED_PIN, LOW);
// Turn OFF the blue LED initially
digitalWrite(LED_BLUE_PIN, LOW);
```

```
  // Pause for 10 seconds (10000ms) to allow system stabilization and user to see init message
  delay(10000);

}

// ========================= MAIN LOOP ====================
// Main loop function: runs repeatedly after setup() completes
void loop() {
  // Get the current time in milliseconds since program start (for non-blocking timing)
  unsigned long currentTime = millis();

  // Check if enough time has passed since last control loop execution (200ms period)
  if (currentTime - last_control_time >= CONTROL_PERIOD_MS) {
    // Measure the current temperature using the NTC thermistor and store in temperatureC
    temperatureC = measur_temp();
    // Apply ON/OFF control logic to adjust the heating element based on temperature vs setpoint
    applyOnOff();
    // Calculate the control error: difference between desired setpoint and measured temperature
    error = setpoint - temperatureC;
    // Update the LED indicators (green/red/blue) based on the current error value
    updateLEDs();
    // Send temperature, setpoint, and control output state to serial port for monitoring/logging
    serialdata();
    // Update the timestamp for the last control loop execution
    last_control_time = currentTime;
  }

  // Check if enough time has passed since last display update (200ms period)
  if (currentTime - last_display_time >= DISPLAY_PERIOD_MS) {
    // Refresh the OLED display with current temperature, setpoint, and error values
    updateDisplay();
    // Update the timestamp for the last display refresh
    last_display_time = currentTime;
  }

}

// ========================= ON/OFF Controller ====================
// Function: Apply simple ON/OFF (bang-bang) control logic for temperature regulation
void applyOnOff() {
   // If measured temperature is at or above the setpoint, turn OFF the heating element
   if (temperatureC >= setpoint) {
    // Set CONTROL_OUTPUT_PIN to LOW to deactivate the relay/heater
    digitalWrite(CONTROL_OUTPUT_PIN, LOW);
    // Update the control output state flag to false (OFF)
```

```
    controlOutputBool = false;
  // Otherwise, if temperature is below setpoint, turn ON the heating element
  } else {
    // Set CONTROL_OUTPUT_PIN to HIGH to activate the relay/heater
    digitalWrite(CONTROL_OUTPUT_PIN, HIGH);
    // Update the control output state flag to true (ON)
    controlOutputBool = true;
  }
}

// ========================= LEDs ====================
// Function: Update LED indicators to visually represent system status based on error
void updateLEDs() {
  // Turn OFF all LEDs initially to ensure only one is lit at a time (mutual exclusion)
  digitalWrite(LED_RED_PIN,   LOW);
  digitalWrite(LED_GREEN_PIN, LOW);
  digitalWrite(LED_BLUE_PIN,  LOW);

  // If the absolute error is within ±0.5°C, temperature is on target: light GREEN LED
  if (fabsf(error) <= 0.5f) {
    digitalWrite(LED_GREEN_PIN, HIGH);  // On target: green LED ON
  // If error is positive and > 0.5°C, temperature is too cold: light RED LED
  } else if (error > 0.5f) {
    digitalWrite(LED_RED_PIN, HIGH);    // Too cold: red LED ON
  // If error is negative and < -0.5°C, temperature is too hot: light BLUE LED
  } else {
    digitalWrite(LED_BLUE_PIN, HIGH);   // Too hot: blue LED ON
  }
}

// ========================= OLED: Normal Display ====================
// Function: Update the OLED display with current temperature, setpoint, and error values
void updateDisplay() {
  // Clear the display buffer to prepare for new content rendering
  display.clearDisplay();
  // Set text color to white for all subsequent text drawing operations
  display.setTextColor(SSD1306_WHITE);

  // Row 1: Display current temperature in large font (size 2x)
  // Set text size to 2x for prominent temperature display
  display.setTextSize(2);
  // Position cursor at top-left corner (column 0, row 0)
  display.setCursor(0, 0);
  // Print the label "T:" to indicate temperature value
  display.print("T:");
  // Print the current temperature value with 1 decimal place precision
```

```
  display.print(temperatureC, 1);
  // Print the degree symbol (ASCII character 247 = °) for temperature units
  display.print((char)247);
  // Print the unit "C" for Celsius
  display.print("C");

  // Row 2: Display setpoint and error values in smaller font (size 1x, default)
  // Reset text size to 1x for secondary information display
  display.setTextSize(1);
  // Position cursor for setpoint display (column 0, row 20)
  display.setCursor(0, 20);
  // Print the label "SP:" to indicate setpoint value
  display.print("SP:");
  // Print the current setpoint value with 1 decimal place precision
  display.print(setpoint, 1);
  // Print the degree symbol (°) for temperature units
  display.print((char)247);
  // Print the unit "C" for Celsius, followed by spacing
  display.print("C  ");
  // Position cursor for error display on next row (column 0, row 40)
  display.setCursor(0, 40);
  // Print the label "Er:" to indicate error value
  display.print("Er:");
  // Print the current error value with 1 decimal place precision
  display.print(error, 1);
  // Print the degree symbol (°) for temperature units
  display.print((char)247);
  // Print the unit "C" for Celsius
  display.print("C");
    // Send the rendered buffer content to the physical OLED display to make it visible
  display.display();
}
//  ===Function: Send temperature, setpoint, and control output state to serial port for monitoring ==
void serialdata()
{
  // Print the current temperature value with 2 decimal places to serial port
  Serial.print(temperatureC, 2);// Fig.A33 blue curve
  // Print a comma delimiter for CSV-style data formatting
  Serial.print(",");
  // Print the current setpoint value with 2 decimal places to serial port
  Serial.print(setpoint, 2); // Fig.A33 red curve
  // Print a comma delimiter for CSV-style data formatting
  Serial.print(",");
  // Print the control output state as "5" (ON) or "0" (OFF), followed by newline
  Serial.println(controlOutputBool ? "5" : "0");  // Fig.A33 green curve
}
```

```
//======================== ISRoutines=========================
// Interrupt Service Routine: Handle DOWN button press to decrease temperature setpoint
void Interrupt_BUTTON_DOWN() {
  // Calculate new setpoint value by subtracting 0.5°C from current setpoint
  float newSetpoint = setpoint - 0.5f;
  // Check if new setpoint is above minimum allowed limit (0.0°C) to prevent invalid values
  if (newSetpoint >= 0.0f) {  // Minimum limit check
    // Update the global setpoint variable with the new decreased value
    setpoint = newSetpoint;
  }
}

// Interrupt Service Routine: Handle UP button press to increase temperature setpoint
void Interrupt_BUTTON_UP() {
  // Calculate new setpoint value by adding 0.5°C to current setpoint
  float newSetpoint = setpoint + 0.5f;
  // Check if new setpoint is below maximum allowed limit (40.0°C) to prevent invalid values
  if (newSetpoint <= 40.0f) {  // Maximum limit check
    // Update the global setpoint variable with the new increased value
    setpoint = newSetpoint;
  }
}

// ============ TEMPERATURE MEASUREMENT (Averaged NTC) ===========
// Function: Measure temperature using NTC thermistor with voltage divider and Steinhart-Hart
//equation
float measur_temp() {
  // Initialize variable to accumulate ADC readings for averaging (reduce noise)
  unsigned long Vo = 0;
  // Loop 200 times to collect multiple ADC samples for averaging
  for(int i = 0; i < 200; i++) {
    // Read analog voltage from pin A0 (NTC voltage divider output) and add to accumulator
    Vo += analogRead(A0);
    // Short delay to allow ADC circuit to settle between readings (improve accuracy)
    delayMicroseconds(10);
  }
  // Calculate the average of 200 ADC readings to reduce measurement noise
  float averageVo = Vo / 200.0f;

  // Calculate the NTC thermistor resistance (R2) using voltage divider formula: R2 = R1 * ((1023/Vo)   - 1)
  float R2 = R1 * (1023.0f / averageVo - 1.0f);
  // Calculate the natural logarithm of R2 for use in Steinhart-Hart equation
  float logR2 = logf(R2);
  // Apply Steinhart-Hart equation to convert resistance to temperature in Kelvin
  float tKelvin = 1.0f / (c1 + c2 * logR2 + c3 * logR2 * logR2 * logR2);
```

```
  // Convert Kelvin to Celsius
  float tCelsius = tKelvin - 273.15f;
  // Constrain the final temperature value to reasonable physical bounds (0°C to 85°C) and return
  return constrain(tCelsius, 0.0f, 85.0f);
}
/*Sketch uses 17670 bytes (54%) of program storage space. Maximum is 32256 bytes.
Global variables use 598 bytes (29%) of dynamic memory, leaving 1450 bytes for local variables. Maximum is 2048 bytes.*/
```

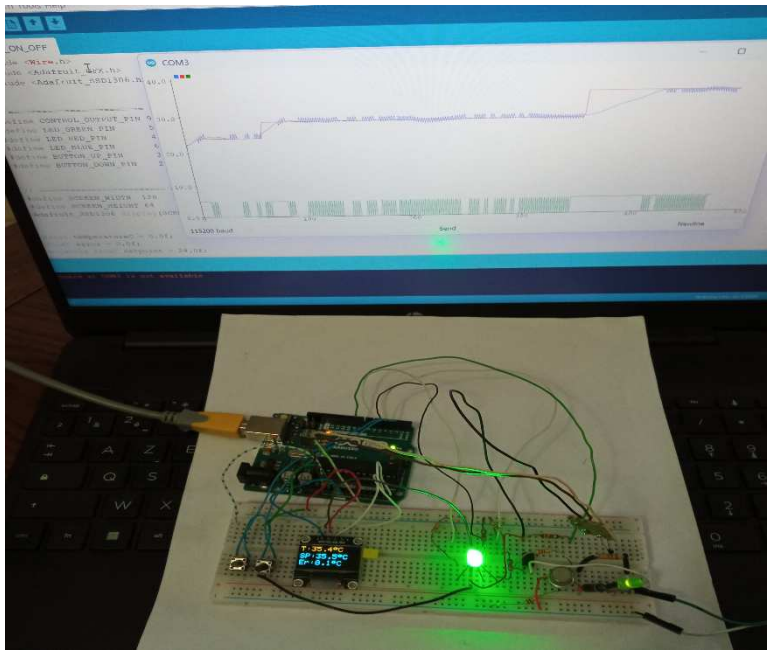


Fig.A31: Practical Setup

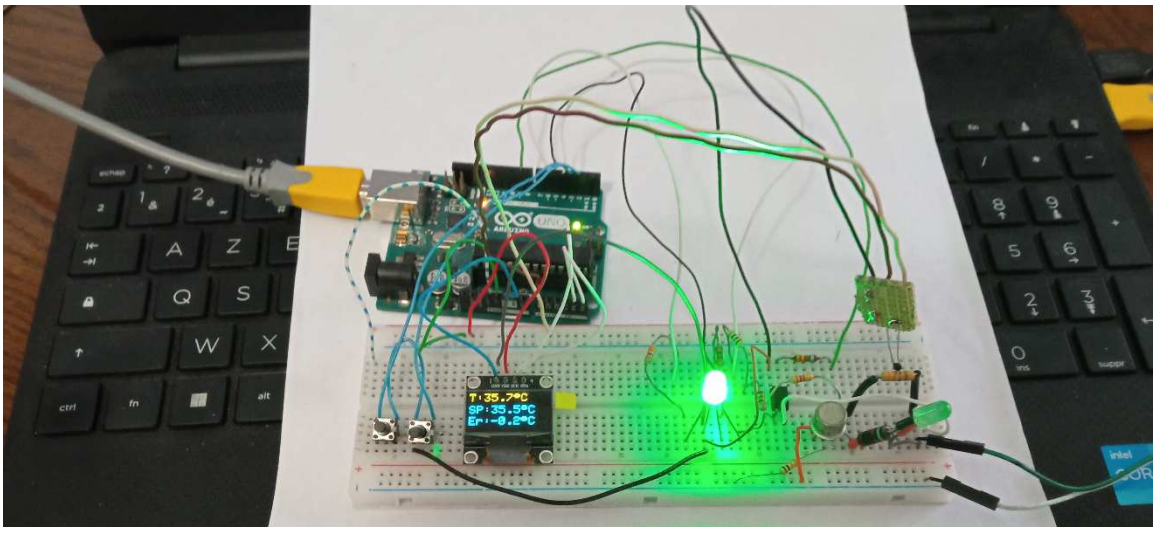


Fig.A32: Experimental Setup of the Temperature Controller on a Breadboard

Fig.A33: Serial Plotter