1 Introduction to Peripherals and Interfacing

Embedded systems are designed to interact with their environment, and this interaction is made possible through peripherals. Peripherals are external devices or components connected to the microcontroller that allow it to sense, process, and act upon inputs and outputs. They include a wide range of devices such as sensors, actuators, and displays, each serving a unique role in enhancing the functionality of the system. Interfacing these peripherals with the microcontroller is a critical aspect of embedded system design, ensuring smooth communication between the system and its surroundings.

The ATmega32 microcontroller, known for its flexibility and extensive peripheral support, is equipped with features like GPIO (General-Purpose Input/Output) pins, ADC (Analog-to-Digital Converter), timers, and PWM (Pulse Width Modulation). These features make it highly suitable for connecting and controlling a variety of peripherals. For example, GPIO pins allow the microcontroller to interact with simple digital devices like switches and LEDs, while the ADC converts analog signals from sensors into digital data that the microcontroller can process. Similarly, PWM enables precise control of actuators like motors and servos.

Interfacing peripherals with a microcontroller like the ATmega32 involves understanding both the hardware connections and the software logic required for communication. This includes setting up the appropriate registers, configuring pins, and writing code to handle data exchange. Each peripheral has unique requirements that dictate how it should be connected and controlled. For instance, while sensors are primarily used to collect data from the environment, actuators are employed to perform actions based on the processed data, and displays are used to present information to users.

As we delve into the details of peripheral interfacing, this chapter will begin with **sensor interfacing**, which is fundamental in most embedded applications. Sensors, acting as the "eyes and ears" of an embedded system, gather critical environmental data such as temperature, humidity, motion, or light. The ATmega32's ADC module plays a key role in interfacing analog sensors, while its GPIO pins simplify the connection of digital sensors.

1.1 Analog-to-Digital Converters (ADC)

Analog-to-digital converters (ADCs) are essential components in data acquisition systems, bridging the gap between the analog world and digital computing. While digital systems, such as microcontrollers, operate using discrete binary values, the physical world operates in a continuous, analog manner. Everyday quantities like temperature, pressure, humidity, and velocity are inherently analog in nature.

To interact with these physical phenomena, we use transducers, commonly referred to as sensors. Transducers convert physical quantities into electrical signals, typically in the form of voltage or current. For instance, temperature sensors produce a voltage proportional to the temperature, while pressure and velocity sensors generate electrical signals corresponding to the force or speed they measure.

Since microcontrollers cannot directly interpret analog signals, an ADC is required to translate these continuous signals into a digital format. The ADC captures the analog voltage or current from a sensor and converts it into a digital number that the microcontroller can read and process. This conversion allows embedded systems to measure, analyze, and respond to changes in the physical environment accurately and efficiently.

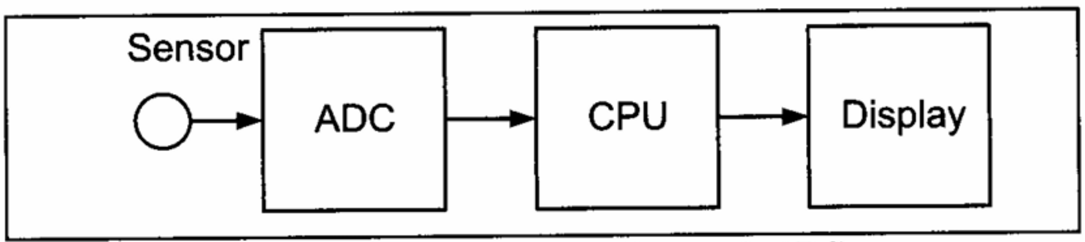


Figure 13-1. Microcontroller Connection to Sensor via ADC

1.1.1 Characteristics of ADC

1. **Resolution**

   The resolution of an Analog-to-Digital Converter (ADC) defines how finely it can divide the input voltage range into discrete digital values. Common resolutions include 8, 10, or 12 bits, with higher resolutions providing greater precision by reducing the step size—the smallest measurable change in input voltage. The step size is determined by the formula:

Step Size $= \frac{V_{ref}}{2^n}$

Where, n is the resolution in bits and V_{ref} *is the reference voltage.*

For instance, an 8-bit ADC with a V_{ref} of 5V divides the input range into 2^8 = 256 steps, resulting in a step size of 5 V/256≈19.53 mV. Similarly, a 10-bit ADC with the same V_{ref} offers a finer step size of 5 V/1024≈4.88 mV. Adjusting V_{ref} changes the step size, enabling better control over ADC precision.

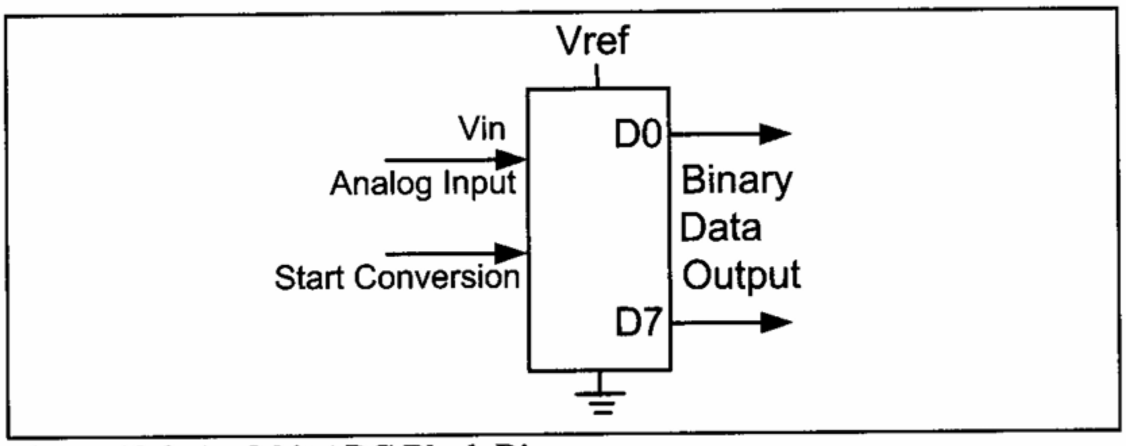


Figure 13-2. An 8-bit ADC Block Diagram

Example: If a temperature sensor outputs a voltage ranging from 0 to 2.56V, setting V_{ref} to 2.56V on an 8-bit ADC yields a step size of 2.56 V/256=10 mV, making it suitable for precise temperature measurements.

Vref (V)	Range (V)	Step Size (mV) (8-bit)	Step Size (mV) (10-bit)
5.00	0 to 5.00	19.53	4.88
4.00	0 to 4.00	15.62	4.00
2.56	0 to 2.56	10.00	2.50
1.024	0 to 1.024	4.00	1.00

2. Conversion Time

Conversion time is the duration the ADC requires to convert an analog signal into its corresponding digital value. This depends on the ADC's clock speed, the method of conversion (e.g., successive approximation or dual slope), and the fabrication technology (MOS or TTL). For real-time applications, such as motion control or audio signal processing, faster conversion times are essential.

Example: If an ADC operates at a clock frequency of 1 MHz and requires 13 clock cycles for one conversion, the conversion time is 13 µs, allowing up to 76,923 conversions per second.

3. **Digital Data Output**

The digital output of an ADC corresponds to its resolution. An 8-bit ADC outputs an 8-bit binary number (ranging from 0 to 255), while a 10-bit ADC outputs a 10-bit number (ranging from 0 to 1023). The output value is related to the input voltage by the formula:

$V_{in} = D_{out} \times Step\ size$

where D_{out} is the digital output in decimal, and Step Size is as defined earlier.

Example: For an 8-bit ADC with V_{ref} = 2.56 V, if the digital output is 128, the corresponding input voltage is 128×10 mV=1.28 .

ADC chips provide data output either serially (bit by bit) or in parallel (all bits simultaneously). The choice depends on the application; parallel output is faster but requires more pins, while serial output is slower but more economical in pin usage. These characteristics enable ADCs to serve as vital components in embedded systems, converting real-world analog signals into digital data for precise and efficient processing.

4. **Start of Conversion (SOC)**

The Start of Conversion (SOC) is the signal that initiates the ADC's process of converting an analog input into a digital value. SOC can be triggered manually by a microcontroller command, periodically by a timer, or in response to an external event. Once the SOC is received, the ADC samples the input signal and generates the corresponding digital output after the conversion time. This mechanism ensures precise and synchronized data acquisition, making SOC a crucial feature for efficient operation in embedded systems.

1.1.2 ADC Features of the ATmega32

- **10-bit resolution**: The ADC can convert analog signals into digital values with 10-bit precision, allowing for 1024 discrete steps.
- **Multiple input channels**: The ATmega32 offers 8 analog input channels, including 7 differential input channels and 2 differential input channels with optional gains of

10x and 200x. This flexibility allows for a wide range of input voltages to be measured accurately.

- **Data registers**: The converted output data is stored in two special function registers: `ADCL` (A/D Result Low) and `ADCH` (A/D Result High). Together, these registers hold 16 bits, with the ADC data occupying the lower 10 bits. The upper 6 bits are unused and can be disregarded in the conversion process.
- **Unused bits**: Since the ADC output is 10 bits wide but the `ADCH:ADCL` registers are 16 bits, there are 6 bits that do not contain useful data. These bits can either be ignored or discarded depending on the specific application requirements.
- **Reference voltage options**: The ADC has three reference voltage options:
  - **AVCC**: Connect the reference voltage to the Vcc supply.
  - **Internal 2.56 V reference**: Provides a stable reference voltage internally within the ATmega32.
  - **External AREF pin**: Allows connection of an external reference voltage for greater flexibility in different applications.
- **Conversion time**: The conversion time of the ADC depends on the clock frequency connected to the XTAL pins (F_{osc}) and the settings of the ADPS bits. These bits control the division factor used by the ADC, affecting how quickly the conversion is completed.

1.1.3 AVR ADC Hardware Considerations

When working with digital logic, small variations in voltage levels usually have no impact on the output. For instance, in TTL logic, any voltage below 0.5 V is detected as a LOW logic level. However, this is not the case when dealing with analog voltages, where small changes can significantly affect the output.

To improve the accuracy of the AVR ADC (Analog-to-Digital Converter) and minimize the effect of supply voltage and V_{ref} variation, several techniques can be used. Two widely used techniques in AVR microcontrollers include:

1. **Decoupling AVCC from VCC**: The AVCC pin supplies power to the analog circuitry of the ADC. To maintain a stable and accurate voltage, it should be decoupled from the VCC (digital power supply) using an inductor and a capacitor,

as shown below. This setup helps filter out noise and fluctuations in the power supply that could affect the ADC readings.

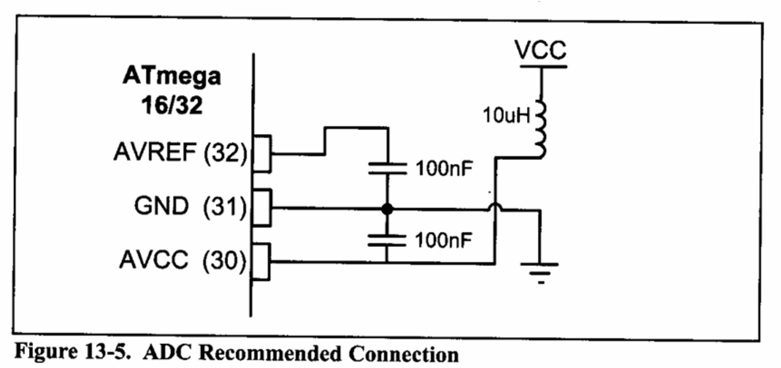


Figure 13-5. ADC Recommended Connection

2. **Connecting a Capacitor Between Vref and GND**: By placing a capacitor between the AVREF pin (reference voltage) and GND (ground), you can stabilize the V_{ref} voltage. This practice reduces fluctuations and increases the precision of the ADC. Figure above illustrates this connection, highlighting how it helps in providing a stable reference voltage for accurate analog-to-digital conversion.

1.2 ADC Programming in the AVR

The ADC in AVR microcontrollers is managed through four main registers. The **ADCH** register holds the high byte, while the **ADCL** register stores the low byte of the ADC conversion result. The **ADCSRA** (ADC Control and Status Register) is used to control and monitor the ADC's operation. The **ADMUX** (ADC Multiplexer Selection Register) configures the reference voltage and selects the input channel for the ADC conversion.

1. ADC Result Alignment: ADCH and ADCL Registers

The AVR microcontroller has a 10-bit ADC, meaning the conversion result is 10 bits long and cannot fit into a single 8-bit register. To accommodate this, the ADC result is stored in two 8-bit registers: **ADCL** (low byte) and **ADCH** (high byte). Out of the 16 bits in these registers, only 10 are used, while the remaining 6 bits are unused. The position of the used bits within the registers is determined by the **ADLAR** bit in the **ADMUX** register.

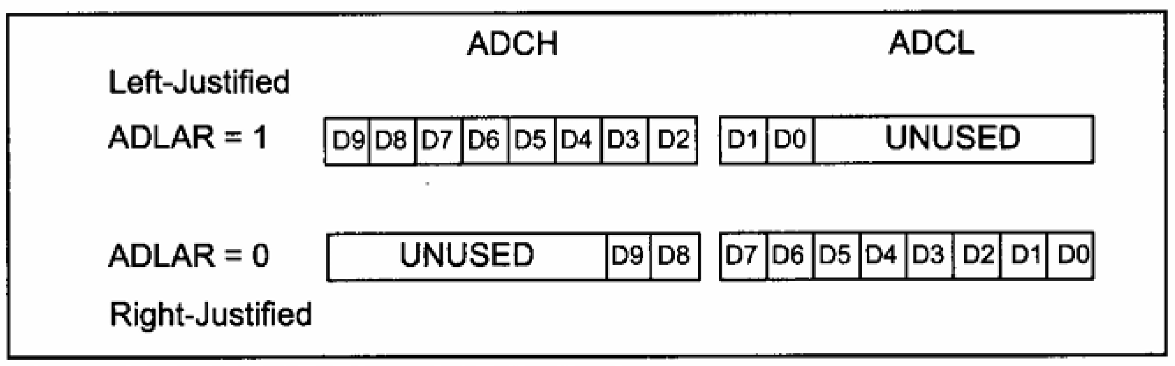
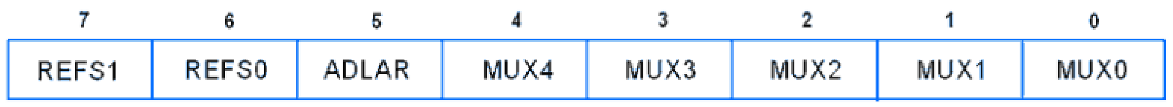


Figure 13-9. ADLAR Bit and ADCx Registers

If the **ADLAR** bit is set to 1, the ADC result is left-justified, meaning the 10 significant bits are aligned to the left, and the unused bits occupy the lower positions in **ADCL**. Conversely, if **ADLAR** is set to 0, the result is right-justified, aligning the significant bits to the right, with unused bits occupying the upper positions in **ADCH**. This behavior allows flexibility in handling the ADC data, but any change to the **ADLAR** bit immediately impacts the alignment of the ADC result in the **ADCL** and **ADCH** registers.

2. ADMUX Register

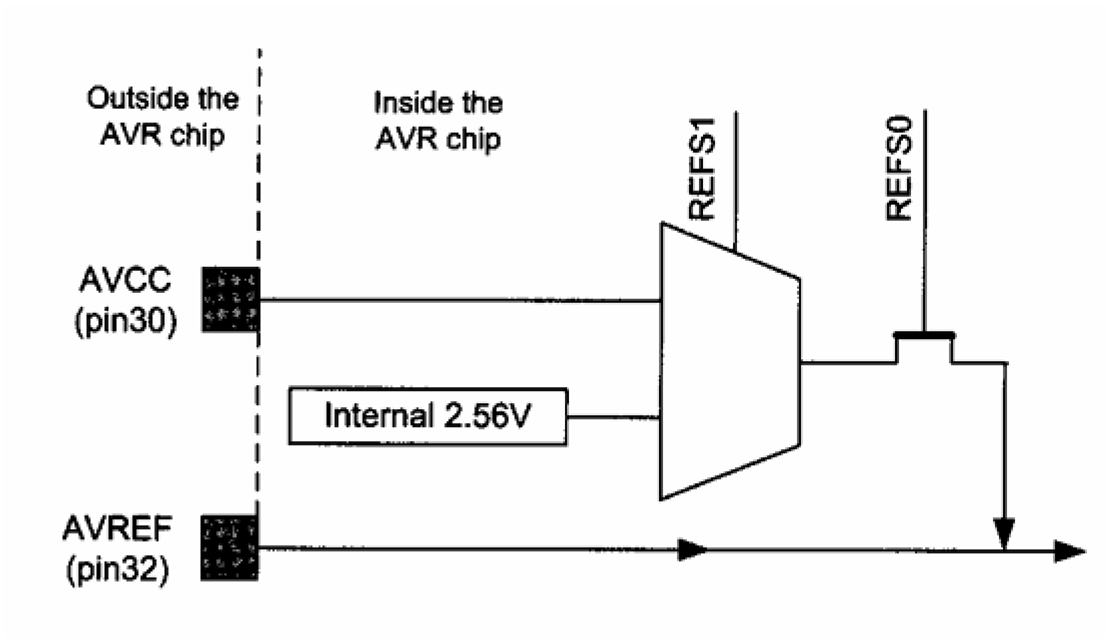
The **ADMUX (ADC Multiplexer Selection) register** is a key component in setting up the ADC (Analog-to-Digital Converter) operation in AVR microcontrollers. It contains various bits that control how the ADC operates, including the reference voltage source and which analog input channel is selected.



7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0

1. Bits 7 and 6 – REFS1 and REFS0 (Reference Voltage Selection)

The REFS1 and REFS0 bits in the ADMUX register are used to select the reference voltage (Vref) for ADC conversion. The reference voltage determines the upper range of the ADC measurement.



The following table shows the available options:

REFS1	REFS0	Reference Voltage (Vref)
0	0	**AREF pin** (External reference)
0	1	**AVCC pin** (Supply voltage, Vcc, 5V)
1	0	Reserved (Not used)
1	1	**Internal 2.56V** (Built-in voltage)

- **AREF Pin** (REFS1 = 0, REFS0 = 0):
    - The reference voltage is provided externally through the **AREF pin**.
    - Ensure no other reference voltage (AVCC or internal) is selected to avoid short circuits.
- **AVCC Pin** (REFS1 = 0, REFS0 = 1):
    - The ADC uses the supply voltage (**Vcc**) as the reference voltage.
    - This option is simple and commonly used when Vcc is stable (e.g., 5V).
- **Internal 2.56V** (REFS1 = 1, REFS0 = 1):
    - The ADC uses a stable built-in 2.56V reference voltage.
    - This is useful for precise and consistent measurements, especially when Vcc fluctuates.

2. **Bit 5 – ADLAR: ADC Left Adjust Result**

The **ADLAR** bit (ADC Left Adjust Result) in the **ADMUX register** controls how the 10-bit ADC result is stored in the **ADCH** (high byte) and **ADCL** (low byte) registers.

When an ADC conversion is performed, the 10-bit result needs to be stored in two 8-bit registers:

- **ADCH**: Stores the upper 8 bits.
- **ADCL**: Stores the lower 2 bits (remaining part of the 10-bit result).

3. Bits 4:0 – MUX4:0: Analog Channel and Gain Selection

The **MUX4:0** bits in the **ADMUX register** are used to select the input channel for the ADC (Analog-to-Digital Converter). These bits allow you to choose which analog input pin will be used for ADC conversion.

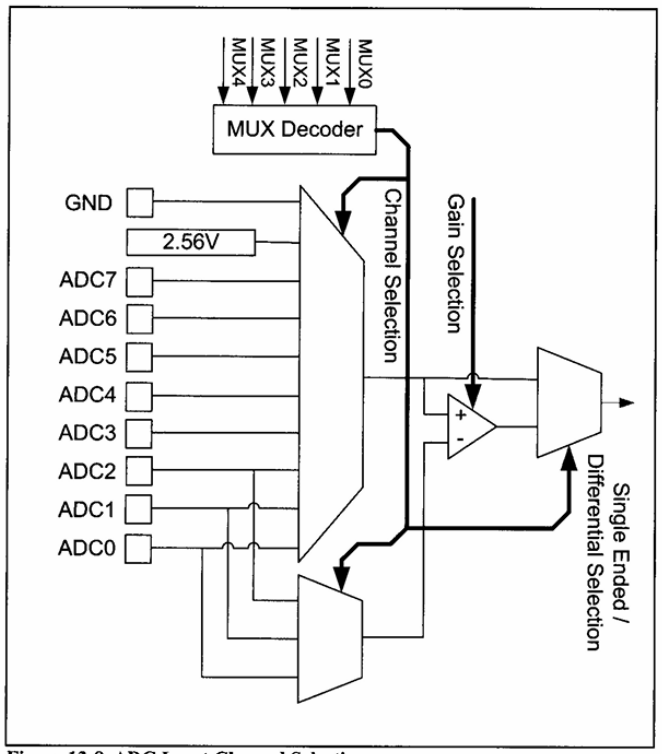


Figure 13-8. ADC Input Channel Selection

Single-Ended Input Channels

When using **single-ended input**, you can select one of the ADC channels (**ADC0 to ADC7**). In this case:

- A single pin is used as the analog input.

- The ground reference for the ADC is the **GND** pin of the AVR microcontroller.

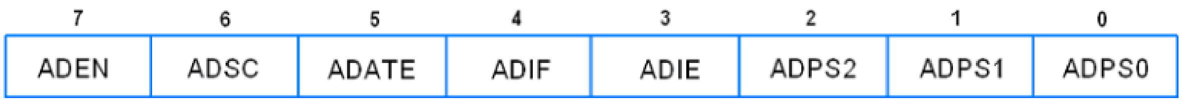
To select a specific ADC channel, set the **MUX4:0** bits to the corresponding channel number.

MUX4:0 Value	Input Channel
00000	ADC0
00001	ADC1
00010	ADC2
00011	ADC3
00100	ADC4
00101	ADC5
00110	ADC6
00111	ADC7

For example, if you want to use **ADC2** as the input channel, set **MUX4:0 = 00010** similarly to use **ADC5** as the input channel, set **MUX4:0 = 00101**.

4. ADCSRA Register: Status and Control

The **ADCSRA (ADC Control and Status Register A)** is used to control and monitor the operation of the ADC (Analog-to-Digital Converter).



7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

1. **Bit 7 – ADEN: ADC Enable**

   Writing one to this bit enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

2. **Bit 6 – ADSC: ADC Start Conversion**

   Writing one to this bit starts the conversion.

3. **Bit 5 – ADATE: ADC Auto Trigger Enable**

   Writing one to this bit, results in Auto Triggering of the ADC is enabled.

4. **Bit 4 – ADIF: ADC Interrupt Flag**

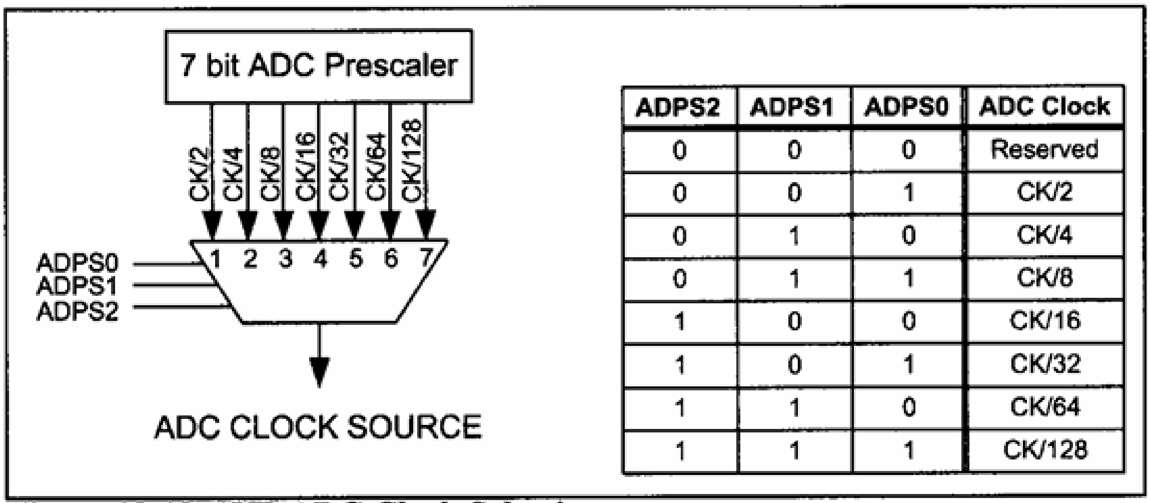
   This bit is set when an ADC conversion completes and the Data Registers are updated.

5. **Bit 3 – ADIE: ADC Interrupt Enable**

   Writing one to this bit, the ADC Conversion Complete Interrupt is activated.

6. **Bits 2 : 0 – ADPS2 : 0: ADC Prescaler Select Bits**

   These bits determine the division factor between the XTAL frequency and the input clock to the ADC



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

Figure 13-12. AVR ADC Clock Selection

Steps to Program the ADC in ATmega16/32:

1. **Enable the ADC Module:**
   - Turn on the ADC by setting the `ADEN` bit in the `ADCSRA` register.
2. **Set the Reference Voltage:**
   - Use the `REFS0` and `REFS1` bits in the `ADMUX` register to select the reference voltage (e.g., AVCC, internal reference, or external reference).
3. **Choose the ADC Input Channel:**
   - Use the `MUX0` to `MUX4` bits in the `ADMUX` register to select the desired ADC channel (e.g., ADC0 to ADC7).
4. **Set the Conversion Speed (Prescaler):**

- Configure the `ADPS0` to `ADPS2` bits in the `ADCSRA` register to set the ADC clock prescaler for optimal conversion speed (50-200 kHz recommended).

5. **Start the ADC Conversion:**
   - Write a 1 to the `ADSC` bit in the `ADCSRA` register to start the conversion.
6. **Wait for the Conversion to Complete:**
   - Monitor the `ADIF` bit in the `ADCSRA` register. It will go HIGH when the conversion is done.
7. **Read the ADC Result:**
   - Read the digital value from the `ADCL` (low byte) and `ADCH` (high byte) registers. Always read `ADCL` first to ensure valid data.
8. **Repeat or Switch Channels:**
   - For another conversion on the same channel, repeat from step 5.
   - For a different channel, go back to step 3.
9. **Disable ADC When Not Needed:**
   - Turn off the ADC by clearing the `ADEN` bit in the `ADCSRA` register to save power.

Programming Example

1. **Program gets data from channel 0 (ADCO) of ADC and displays the result on Port C and Port D.**

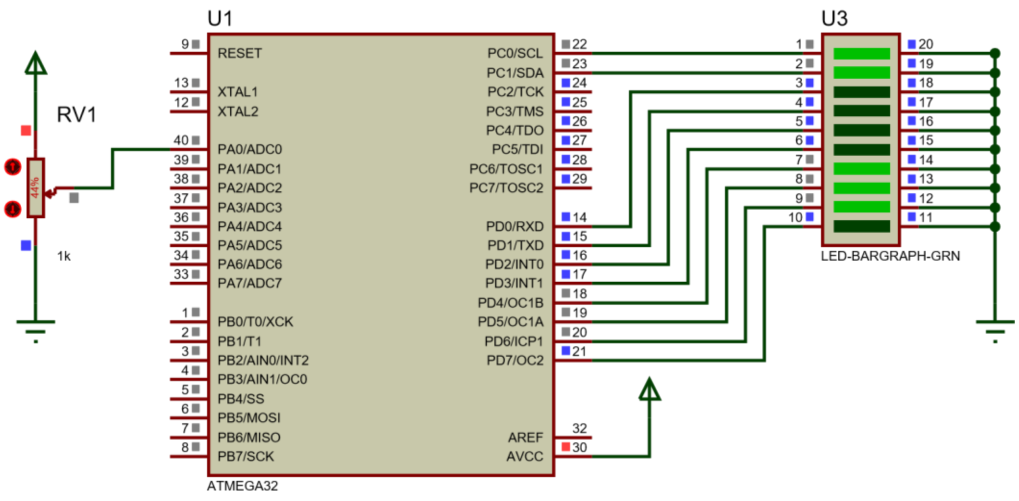
   Code

```
/*
 * GccApplication5.c
 *
 * Created: 12/27/2024 7:05:49 AM
 * Author : Deepesh
 */

#include <avr/io.h> // Standard AVR header

int main(void) {
       // Configure ports
       DDRC = 0xFF;  // Make Port B an output
       DDRD = 0xFF;  // Make Port C an output
       DDRA = 0x00;  // Make Port A an input for ADC input

       // Configure ADC
       ADCSRA = 0x87; // Enable ADC and set prescaler to ck/128
       ADMUX = 0xC0;  // Set Vref to 2.56V (internal), ADC0 as single-ended input,
       // and right-justify the ADC result

       while (1) {
```

```
                ADCSRA |= (1 << ADSC);       // Start ADC conversion

                while ((ADCSRA & (1 << ADSC))); // Wait for conversion to finish

                PORTD = ADCL; // Send low byte of ADC result to Port C
                PORTC = ADCH; // Send high byte of ADC result to Port B
        }

        return 0;
}
```

2. Program gets data from channel 0 (ADCO) of ADC and displays the result UART

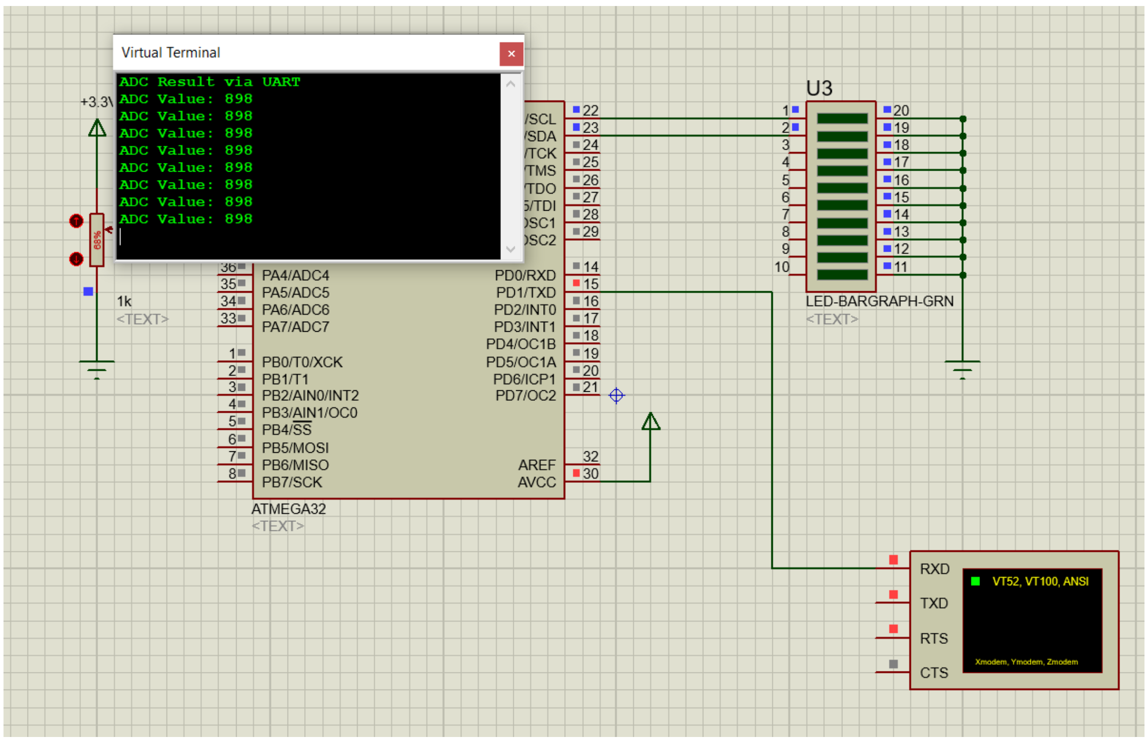
Circuit Diagram



Code

```
/*
 * ADC Result via UART
 * Author: Deepesh
 * Date: 12/27/2024
 */

#define F_CPU 8000000UL // Define CPU speed as 8 MHz

#include <avr/io.h>      // Standard AVR header
#include <util/delay.h> // Delay utility for timing functions

// Function to initialize UART with a given baud rate
void UART_Init(unsigned long baud_rate) {
    unsigned int ubrr = (F_CPU / (16UL * baud_rate)) - 1; // Calculate UBRR value
    UBRRH = (unsigned char)(ubrr >> 8);   // Set upper byte of UBRR
    UBRRL = (unsigned char)ubrr;          // Set lower byte of UBRR
    UCSRB = (1 << RXEN) | (1 << TXEN);    // Enable UART transmitter and receiver
    UCSRC = (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Set frame format: 8 data
bits, 1 stop bit
}
```

```c
// Function to transmit a character over UART
void UART_TxChar(char data) {
    while (!(UCSRA & (1 << UDRE))); // Wait until transmit buffer is empty
    UDR = data;                     // Send the data
}

// Function to send a string over UART
void UART_SendString(const char *str) {
    while (*str) {
        UART_TxChar(*str++); // Send each character until null terminator
    }
}

// Function to initialize ADC
void ADC_Init() {
    DDRA = 0x00;        // Set Port A as input for ADC
    ADCSRA = 0x87;      // Enable ADC and set prescaler to ck/128
    ADMUX = 0xC0;       // Set Vref to 2.56V (internal) and ADC0 as single-ended input
}

// Function to read ADC value
int ADC_Read() {
    int result;
    ADCSRA |= (1 << ADSC); // Start ADC conversion
    while (ADCSRA & (1 << ADSC)); // Wait for conversion to complete
    result = ADCL;                // Read low byte
    result |= (ADCH << 8);        // Combine high byte with low byte
    return result;                // Return the ADC result
}

int main(void) {
    char buffer[10];          // Buffer to store ADC result as string
    UART_Init(9600);          // Initialize UART with 9600 baud rate
    ADC_Init();               // Initialize ADC

    UART_SendString("ADC Result via UART\r\n"); // Send a welcome message

    while (1) {
        int adc_value = ADC_Read();          // Read ADC value
        itoa(adc_value, buffer, 10);         // Convert ADC value to string
        UART_SendString("ADC Value: ");      // Send label
        UART_SendString(buffer);             // Send ADC value
        UART_SendString("\r\n");             // Send newline
        _delay_ms(1000);                     // Delay for readability
    }

    return 0;
}
```



```
ADC Result via UART
ADC Value: 898
ADC Value: 898
ADC Value: 898
ADC Value: 898
ADC Value: 898
ADC Value: 898
ADC Value: 898
ADC Value: 898
```

1.3 Interfacing Temperature Sensor (LM34/35)

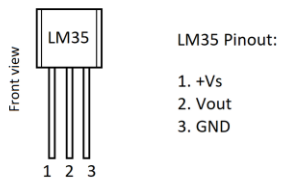
The **LM35** is a commonly used precision temperature sensor designed for measuring temperatures in a variety of applications. It provides a linear output voltage that is directly proportional to the temperature in **Celsius**.

Key Features of LM35:

- **Output Voltage**: The LM35 provides an analog output voltage that is **10 mV/°C** (millivolts per degree Celsius), making it easy to calculate the temperature.
- **Temperature Range**: The LM35 operates over a wide temperature range of **-55°C to +150°C**.
- **Low Power Consumption**: It consumes very little power, making it ideal for battery-powered systems.
- **Accuracy**: The LM35 is accurate to within ±0.5°C over the full temperature range.
- **Linear Output**: The LM35's output voltage changes linearly with temperature, which simplifies the design of temperature measurement systems.
- **Wide Supply Voltage**: The LM35 operates with a supply voltage between **4V and 30V**, making it adaptable to many systems.

LM35 Pinout and Package Details

The LM35 typically comes in an 3-pin **TO-92** or **TO-220** package. Below is the pin configuration for the LM35:



- **Pin 1**: **Vcc** – Power supply input (typically 5V or higher).
- **Pin 2**: **Vout** – Analog output, which gives the temperature-dependent voltage (10 mV/°C).
- **Pin 3**: **GND** – Ground connection.

ADC Step Size and Scaling

The **step size** of the ADC is determined by the reference voltage and the ADC resolution. The **ATmega32** microcontroller features a **10-bit ADC**, which means the ADC resolution is **1024 steps** (from 0 to 1023) for a reference voltage of 5V.

Step Size Calculation:

- If the reference voltage is set to 2.56 V (a commonly used internal reference voltage), the step size can be calculated as follows:

$$Step\ Size = \frac{\text{Reference Voltage}}{1024} = 2.5\,\text{mV per step}$$

The LM35 outputs **10 mV per degree Celsius**, which is much larger than the **2.5 mV** per step of the ADC. This means the ADC reading will be scaled. The digital value from the ADC will need to be divided by **4** to get the correct temperature reading.

Example of Temperature Conversion

Let’s go through an example to see how the **LM35** output is converted into an ADC value and temperature:

1. **For 1°C**:

   The LM35 will output **10 mV**. Since the ADC step size is **2.5 mV**, the ADC value will be:

$$\text{ADC Value} = \frac{10\ \text{mV}}{2.5mV/step} = 4steps$$

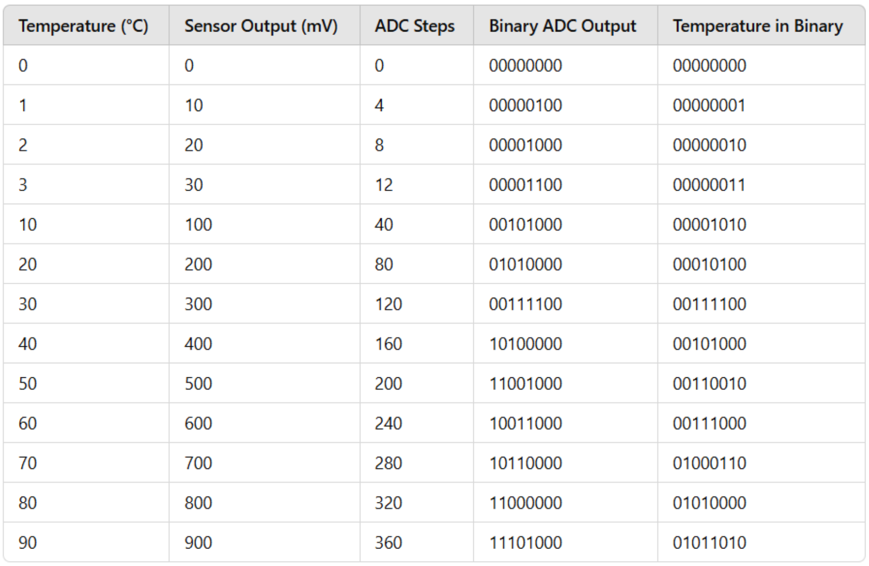
2. **For 2°C**:

   The LM35 will output **20 mV**.The ADC value will be:

$$\text{ADC Value} = \frac{20\ \text{mV}}{2.5mV/step} = 8\ steps$$

This scaling process continues for higher temperatures.

Here's a table showing how the LM35 sensor output corresponds to the ADC value with a **2.56 V** reference voltage:



Temperature (°C)	Sensor Output (mV)	ADC Steps	Binary ADC Output	Temperature in Binary
0	0	0	00000000	00000000
1	10	4	00000100	00000001
2	20	8	00001000	00000010
3	30	12	00001100	00000011
10	100	40	00101000	00001010
20	200	80	01010000	00010100
30	300	120	00111100	00111100
40	400	160	10100000	00101000
50	500	200	11001000	00110010
60	600	240	10011000	00111000
70	700	280	10110000	01000110
80	800	320	11000000	01010000
90	900	360	11101000	01011010

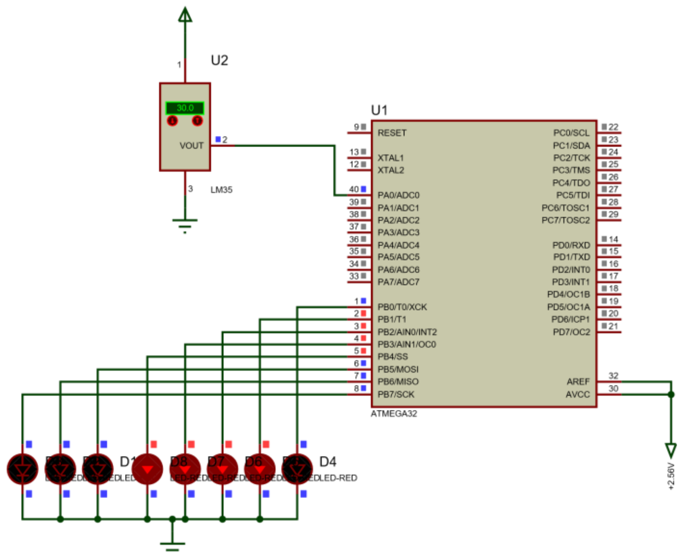
Program to Read Temperature Sensor and display to Port D

Code

```c
/*
 * GccApplication7.c
 *
 * Created: 12/29/2024 6:56:57 PM
 * Author : Deepesh
 */
#define F_CPU 8000000UL // Define CPU speed as 8 MHz
#include <avr/io.h>   //standard AVR header
int main (void)
```

```
{
       DDRB=0xFF;     //make Port B an output
       DDRA=0x00;     //make Port A an input for ADC input
       ADCSRA =0x87; //make ADC enable and select ck/128
       ADMUX = 0xE0;    //2 .56 V Vref and ADCO single—ended  data will be left-
justified
       while (1) {
              ADCSRA |= (1<<ADSC); // Start conversion
              while ( (ADCSRA& (1<<ADIF))==0); //wait for end of conversion
              PORTB = ADCH; //give the high byte to PORTB
       }
       return 0;

}
```

Circuit Diagram

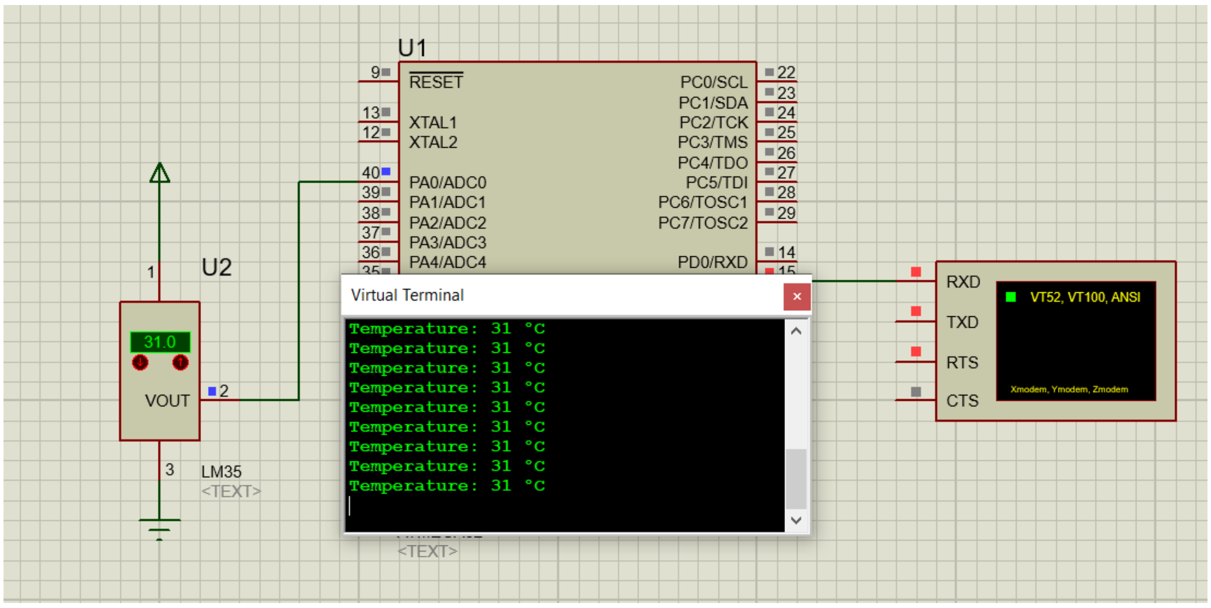


Program to Read LM35 temperature Sensor and Send its reading to UART at baud rate of 9600.

Code:

```
#define F_CPU 8000000UL // Define CPU speed as 8 MHz

#include <avr/io.h>      // Standard AVR header
#include <util/delay.h> // Delay utility for timing functions

// Function to initialize UART with a given baud rate
void UART_Init(unsigned long baud_rate) {
        unsigned int ubrr = (F_CPU / (16UL * baud_rate)) - 1; // Calculate UBRR value
        UBRRH = (unsigned char)(ubrr >> 8);  // Set upper byte of UBRR
        UBRRL = (unsigned char)ubrr;         // Set lower byte of UBRR
        UCSRB = (1 << RXEN) | (1 << TXEN);   // Enable UART transmitter and receiver
        UCSRC = (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1); // Set frame format: 8 data bits, 1 stop bit
}
```

```
// Function to transmit a character over UART
void UART_TxChar(char data) {
        while (!(UCSRA & (1 << UDRE))); // Wait until transmit buffer is empty
        UDR = data;                  // Send the data
}

// Function to send a string over UART
void UART_SendString(const char *str) {
        while (*str) {
                UART_TxChar(*str++); // Send each character until null terminator
        }
}

// Function to initialize ADC
void ADC_Init() {
        DDRA = 0x00;       // Set Port A as input for ADC
        ADCSRA = 0x87;     // Enable ADC and set prescaler to ck/128
        ADMUX = 0xC0;      // Set Vref to 2.56V (internal) and ADC0 as single-ended input
}

// Function to read ADC value
int ADC_Read() {
        int result;
        ADCSRA |= (1 << ADSC); // Start ADC conversion
        while (ADCSRA & (1 << ADSC)); // Wait for conversion to complete
        result = ADCL;              // Read low byte
        result |= (ADCH << 8);       // Combine high byte with low byte
        return result;              // Return the ADC result
}

int main(void) {
        char buffer[10];         // Buffer to store string data
        UART_Init(9600);         // Initialize UART with 9600 baud rate
        ADC_Init();              // Initialize ADC

        UART_SendString("LM35 Temperature Sensor via UART\r\n"); // Send a welcome message

        while (1) {
                int adc_value = ADC_Read();          // Read ADC value
                int temperature = adc_value / 4;     // Convert ADC value to temperature (°C)

                // Convert temperature to string
                itoa(temperature, buffer, 10);
                UART_SendString("Temperature: ");    // Send label
                UART_SendString(buffer);             // Send temperature value
                UART_SendString(" °C\r\n");          // Send unit and newline

                _delay_ms(1000);                     // Delay for readability
        }

        return 0;
}
```



```
Temperature: 31 °C
Temperature: 31 °C
Temperature: 31 °C
Temperature: 31 °C
Temperature: 31 °C
Temperature: 31 °C
Temperature: 31 °C
Temperature: 31 °C
Temperature: 31 °C
```

2 Actuator Interfacing

In the world of embedded systems, **actuators** are essential components that bridge the gap between digital control systems and the physical world. They are devices that perform actions or produce physical changes in response to commands from a microcontroller or processor. Actuators act as the **output** of a system, converting electrical signals into mechanical motion, force, or action. This ability to produce motion or perform tasks makes actuators a crucial part of automation, robotics, and control systems.

To understand their role better, we can think of an embedded system as having three major parts:

- **Sensors** that collect data and provide input.
- **Controllers** (microcontrollers or processors) that process the data and decide on actions.
- **Actuators** that execute those actions by interacting with the physical environment.

For example, in a smart home system, a **temperature sensor** detects room temperature and sends the data to a microcontroller. Based on the sensor reading, the microcontroller commands an actuator (such as a fan motor or thermostat relay) to turn on or off, thereby adjusting the temperature. In robotics, actuators control **motors** to move wheels, robotic arms, or joints, translating digital commands into physical actions.

Actuators come in various forms, such as **DC motors**, **stepper motors**, **servo motors**, **solenoids**, and **relays**, depending on their purpose and the nature of the task they perform. Selecting the right actuator depends on factors like precision, torque, speed, and the type of motion required (linear or rotational).

2.1 DC Motor Introduction

A **Direct Current (DC) motor** is a commonly used actuator that converts electrical energy into mechanical motion. It operates on a simple principle: when a voltage is applied across its two terminals, the motor produces rotational motion. A DC motor has two leads, typically referred to as **positive (+)** and **negative (–)**. When connected to a DC voltage source, the motor rotates in one direction. By reversing the polarity of the voltage, the direction of rotation is reversed. This behavior makes DC motors easy to experiment with and integrate into embedded systems. DC motors are widely used in applications such as robotics, computer cooling fans, toys, and industrial machines due to their simplicity, cost-effectiveness, and ease of control. For instance, small fans used to cool CPUs in computers operate using DC motors. Unlike stepper motors, which move in fixed steps, DC motors provide continuous rotation, making them ideal for applications that require smooth motion. The speed of a DC motor is typically specified in revolutions per minute (**RPM**), and it depends on factors such as load, applied voltage, and current.

2.1.1 Controlling the Rotation of a DC Motor

The rotation of a DC motor can be controlled in two ways: **unidirectional control** and **bidirectional control**.

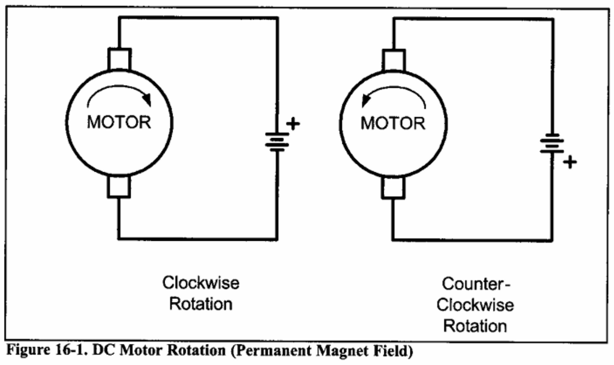


Figure 16-1. DC Motor Rotation (Permanent Magnet Field)

In **unidirectional control**, the motor rotates in a single direction (clockwise or counterclockwise), depending on the polarity of the connected power supply. This type of control is straightforward and is commonly used in applications where the direction of motion does not need to change, such as cooling fans, simple conveyor belts, or drills.

In **bidirectional control**, the motor's direction can be reversed. This can be achieved by changing the polarity of the voltage applied to the motor. Two common methods for bidirectional control are:

- **Relays**: Relays can be used to switch the polarity of the power supply, allowing the motor to rotate in both clockwise and counterclockwise directions.
- **H-Bridge Circuit**: An H-Bridge is a more efficient and popular method for controlling motor direction. It consists of a combination of transistors or MOSFETs that can reverse the polarity of the voltage applied to the motor, enabling bidirectional rotation. H-Bridge circuits are widely used in embedded systems and motor driver modules.

By controlling the voltage, current, and polarity, the speed and direction of the DC motor can be adjusted as required.

2.1.2 H-Bridge Circuit for DC Motor Control

The **H-Bridge** is an essential circuit used for controlling the direction of rotation of a DC motor. It allows bidirectional motor control by altering the polarity of the voltage applied across the motor terminals. The name “H-Bridge” comes from the physical arrangement of its components, where four switches (or transistors/MOSFETs) are configured to resemble the shape of the letter **H**. The motor is connected at the center, forming the crossbar of the "H." H-Bridge is widely used in robotics, motor control systems, and automation projects where the motor needs to move both clockwise (CW) and counterclockwise (CCW). By selectively turning switches ON and OFF, the direction of current flow through the motor is controlled, reversing its rotation

2.1.2.1 How It Works

The H-Bridge consists of **four switches**: **Switch 1 (S1)**, **Switch 2 (S2)**, **Switch 3 (S3)**, and **Switch 4 (S4)**. By controlling the ON/OFF states of these switches, the polarity of the voltage across the motor is reversed. Below is the working principle:

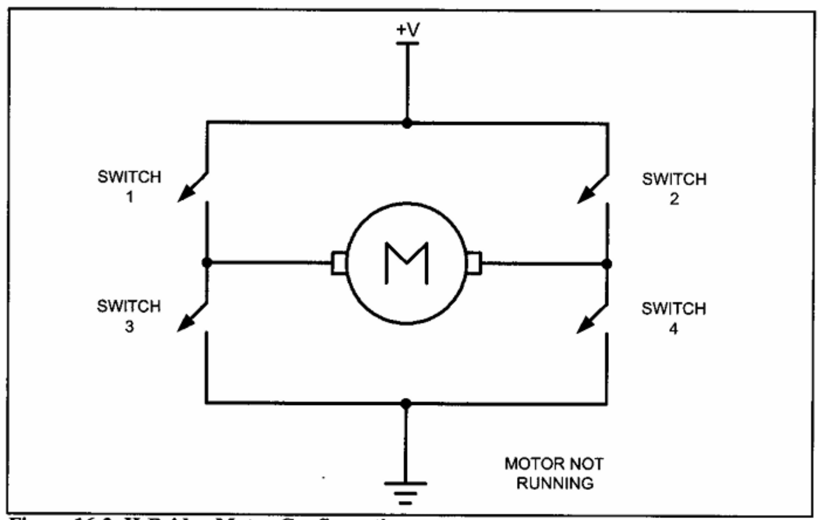


Figure 16-2. H-Bridge Motor Configuration

- **Clockwise Rotation (CW):**
    - **S1 ON**, **S4 ON**, while **S2 OFF**, **S3 OFF**.
    - Current flows from the power supply through **S1**, across the motor (left to right), and exits via **S4**. This polarity causes the motor to rotate in the clockwise direction.
- **Counterclockwise Rotation (CCW):**
    - **S2 ON**, **S3 ON**, while **S1 OFF**, **S4 OFF**.
    - Current flows from the power supply through **S2**, across the motor (right to left), and exits via **S3**. The reversed polarity causes the motor to rotate in the counterclockwise direction.
- **Stopping the Motor:**
    - **All switches OFF**: Disconnects the motor, stopping it naturally.

In practical applications, manually implementing an H-Bridge circuit using discrete switches or transistors can be complex and prone to errors. To simplify the process, motor driver ICs like **L293D** and **L298N** are commonly used. These integrated circuits incorporate the H-Bridge configuration along with essential protection features, such as fly back diodes, current limiting, and thermal protection. Motor driver ICs are designed to interface directly with microcontrollers, allowing easy control of DC motors in both directions with minimal external components.

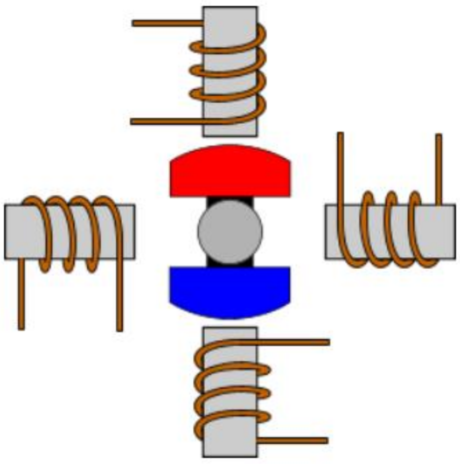
2.1.3 DC Motor Direction Control

2.1.4 Stepper Motor

A stepper motor is a type of electromechanical device that converts electrical signals into mechanical motion. It is specifically designed to perform precise movements, known as steps, to achieve accurate positioning. The motion in a stepper motor is generated by using a magnetic field created by coils and sensed by magnets. When one of the coils is energized, it produces a magnetic field. If the energy is supplied in a cyclic manner through input pulses, the magnetic field continuously changes. A magnet placed within this varying magnetic field adjusts its position to reach its lowest energy state, or equilibrium. This adjustment causes the rotor to move step by step, resulting in mechanical motion. The stepper motor consists of two main parts: the stator and the rotor. The stator is the stationary part that holds the coils, which are energized in cycles to create the magnetic field. The rotor, made of either ferromagnetic material or permanent magnets, is the moving part that interacts with the magnetic field to produce motion. This fundamental design enables stepper motors to provide precise and controlled movements, making them ideal for applications requiring accurate positioning.

2.1.4.1 Basic Operation

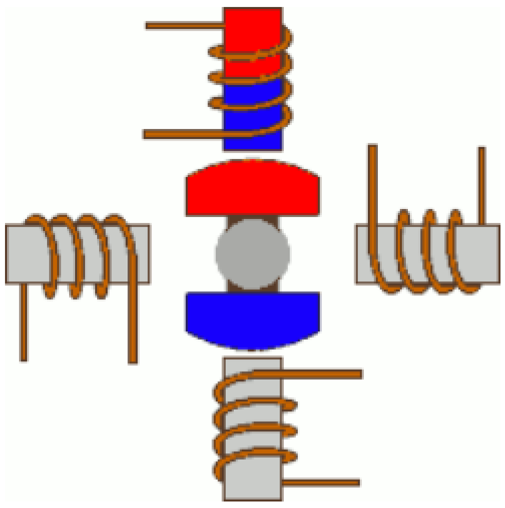
In a basic stepper motor, four coils are fixed on the stator at 90° intervals. These coils are energized sequentially, one after the other, to generate a rotating magnetic field. The rotor, carrying permanent magnets, aligns itself with the magnetic field of the energized coil, causing it to move by a discrete step, typically 90° in this basic design. The direction of the rotor's rotation depends on the order in which the coils are activated.

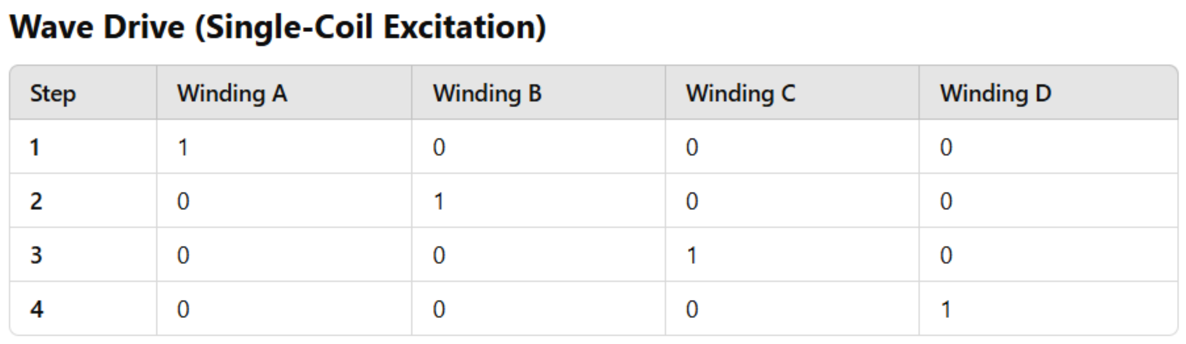


2.1.4.2 Driving Modes in Stepper Motors

1. Wave Drive (Single-Coil Excitation)

In the **Wave Drive** mode, only one coil is energized at a time. This method is simple and consumes minimal power, making it suitable for low-load applications where energy efficiency is critical. However, it provides less than half the nominal torque of the motor, limiting its ability to handle heavy loads. With four coils, the rotor completes one full revolution in four steps, as each coil is energized sequentially.



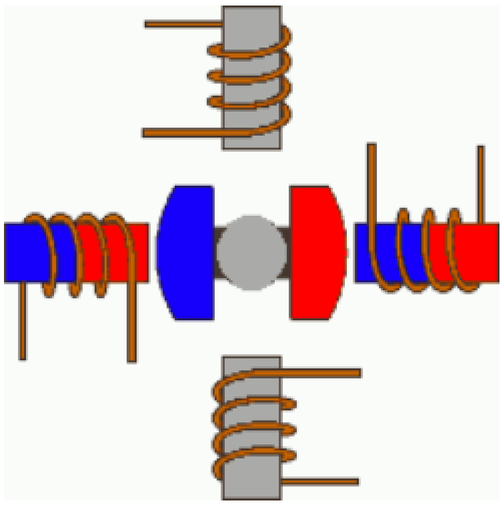


Wave Drive (Single-Coil Excitation)

Step	Winding A	Winding B	Winding C	Winding D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

2. Full Step Drive

The **Full Step Drive** mode energizes pairs of coils simultaneously, resulting in stronger magnetic fields. This produces the full nominal torque of the motor, making it suitable for applications requiring higher load capacity. However, this mode requires double the voltage or current compared to the Wave Drive. Like the Wave Drive, the rotor completes one full revolution in four steps.



Full Step Drive (Two-Coil Excitation)

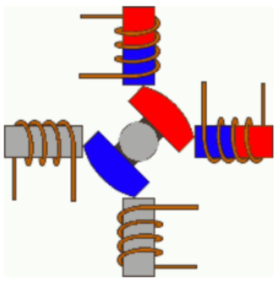
Step	Winding A	Winding B	Winding C	Winding D
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1
4	1	0	0	1

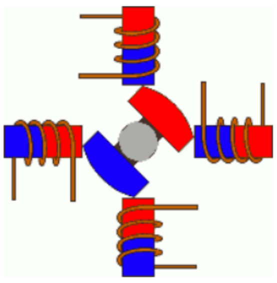
3. Half Stepping

The **Half Stepping** mode combines single-coil and two-coil excitation to achieve finer positional accuracy. This mode alternates between energizing one coil and two coils simultaneously, effectively halving the step size. As a result, the rotor completes one full revolution in eight steps. This method provides moderate torque and is commonly used in applications requiring precise positioning without hardware modifications.

Half Stepping

Step	Winding A	Winding B	Winding C	Winding D
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1


Single-Coil excitation


Two-Coil excitation

2.1.4.3 Step Size in Stepper Motors

The step size of a stepper motor is the smallest angular movement the motor can make in one step. It is determined by the internal construction of the motor, specifically the number of teeth on the rotor and the stator, as well as the drive mode used (e.g., full step or half step). The step size is measured in degrees and is critical in applications requiring precise positioning.

The formula for calculating the step angle (θ) is:

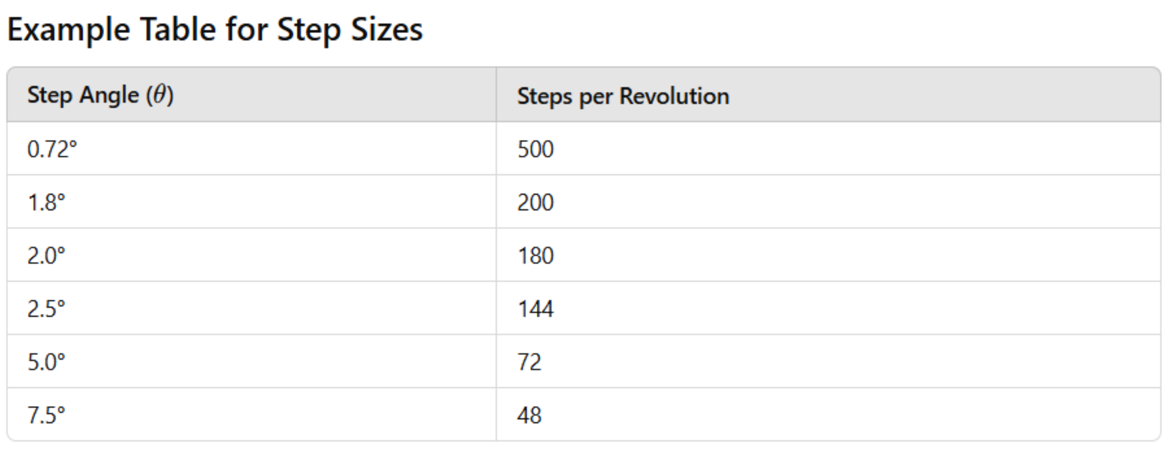
$$\theta = \frac{360^0}{N \times P}$$

Where,

- N = Number of teeth on the rotor
- P = Number of phases in the motor

Using this formula, different stepper motors can have varying step sizes. The step size also directly impacts the steps per revolution, which is calculated as:

$$\text{Steps per Revolution} = \frac{360^0}{Step\ Angle}$$



Example Table for Step Sizes

Step Angle (θ)	Steps per Revolution
0.72°	500
1.8°	200
2.0°	180
2.5°	144
5.0°	72
7.5°	48

Smaller step angles result in higher precision, making them ideal for applications such as CNC machines, 3D printers, and robotics. However, these motors often require more complex drive circuitry to achieve such accuracy.

2.1.5 Programming Uni-Polar Stepper Motor using wave drive

The wave drive of a unipolar stepper motor is a simple driving method where one winding (or coil) is energized at a time. This method results in less torque compared to other driving methods, but it reduces power consumption and simplifies control.

Below is a table representing the wave drive sequence for clockwise (CW) and counterclockwise (CCW) rotation of a unipolar stepper motor.

	A1	A2	B1	B2
Step 1	1	0	0	0
Step 2	0	1	0	0
Step 3	0	0	1	0
Step 4	0	0	0	1

Figure 2-1: For counter clockwise

	A1	A2	B1	B2
Step 1	0	0	0	1
Step 2	0	0	1	0
Step 3	0	1	0	0
Step 4	1	0	0	0

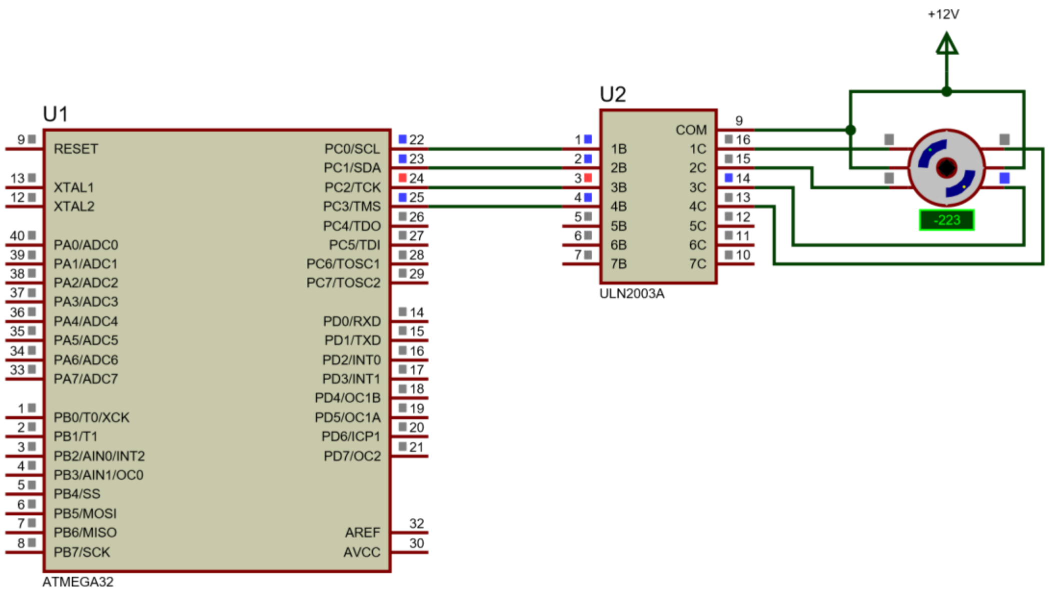
Figure 2-2: For clockwise

Code:

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#define stepTime 100 // Step delay in milliseconds

// Function to rotate the stepper motor in the clockwise direction
void stepForward() {
       PORTC = 0x08; // Activate Winding D
       _delay_ms(stepTime);
       PORTC = 0x04; // Activate Winding C
       _delay_ms(stepTime);
       PORTC = 0x02; // Activate Winding B
       _delay_ms(stepTime);
       PORTC = 0x01; // Activate Winding A
       _delay_ms(stepTime);
}

// Function to rotate the stepper motor in the anticlockwise direction
void stepBackward() {
       PORTC = 0x01; // Activate Winding A
```

```
        _delay_ms(stepTime);
        PORTC = 0x02; // Activate Winding B
        _delay_ms(stepTime);
        PORTC = 0x04; // Activate Winding C
        _delay_ms(stepTime);
        PORTC = 0x08; // Activate Winding D
        _delay_ms(stepTime);
}



int main(void) {
        DDRC = 0x0F; // Set PC0 to PC3 as output for stepper motor control
        PORTC = 0x00; // Clear PORTC to ensure all windings are off initially

        while (1) {
                // Call the desired function for rotation
                //stepForward(); // Uncomment for clockwise rotation
                stepBackward(); // Uncomment for anticlockwise rotation
        }
        return 0;
}
```

Circuit Diagram:



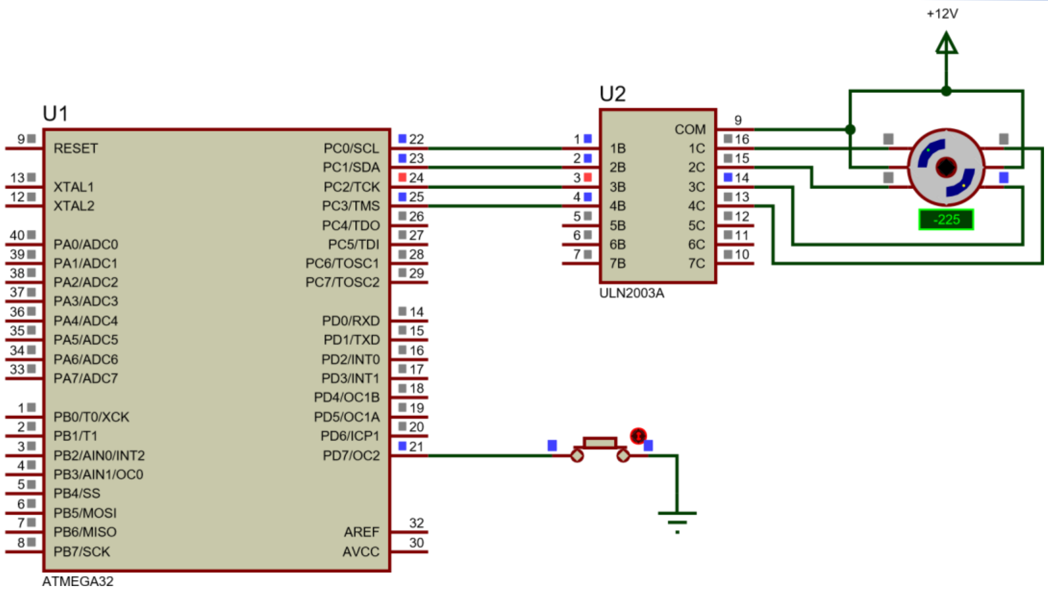
Program to change direction of Stepper Motor based on switch pressed. If switched is pressed the stepper motor should turn counter-clockwise else it should turn clockwise

Code:

```
#include <avr/io.h>
#define F_CPU 16000000UL
#include "util/delay.h"
```

```
#define stepTime 50 // Step delay in milliseconds

// Function to rotate the stepper motor in the clockwise direction
void stepForward() {
       PORTC = 0x08; // Activate Winding D
       _delay_ms(stepTime);
       PORTC = 0x04; // Activate Winding C
       _delay_ms(stepTime);
       PORTC = 0x02; // Activate Winding B
       _delay_ms(stepTime);
       PORTC = 0x01; // Activate Winding A
       _delay_ms(stepTime);
}

// Function to rotate the stepper motor in the anticlockwise direction
void stepBackward() {
       PORTC = 0x01; // Activate Winding A
       _delay_ms(stepTime);
       PORTC = 0x02; // Activate Winding B
       _delay_ms(stepTime);
       PORTC = 0x04; // Activate Winding C
       _delay_ms(stepTime);
       PORTC = 0x08; // Activate Winding D
       _delay_ms(stepTime);
}

int main(void) {
       DDRC = 0x0F; // Set PC0 to PC3 as output for stepper motor control
       PORTC = 0x00; // Clear PORTC to ensure all windings are off initially

       DDRD &= ~(1 << PD7); // Set PD7 as input for switch
       PORTD |= (1 << PD7); // Enable pull-up resistor on PD7

       while (1) {
              if (PIND & (1 << PD7)) {
                     // If the switch is not pressed (PD7 is HIGH due to pull-up)
                     stepForward(); // Rotate clockwise
                     } else {
                     // If the switch is pressed (PD7 is LOW)
                     stepBackward(); // Rotate anticlockwise
              }
       }
}
```

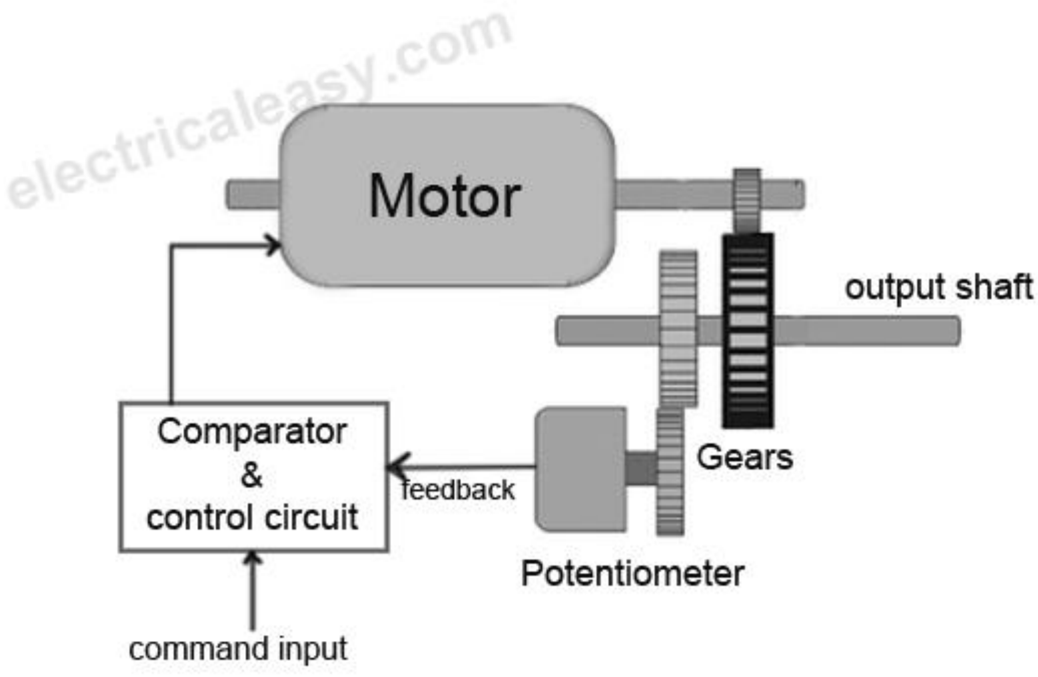
Circuit Diagram:



2.2 Servo Motor

A **servo motor** is a rotary actuator used for precise control of angular position, speed, and acceleration. Unlike stepper motors, servo motors operate with continuous rotation and depend on a feedback system to ensure accurate positioning. They are widely used in robotics, industrial automation, and systems requiring precise motion control.

Servo motors consist of key components such as a motor (DC or AC), a feedback device (potentiometer or encoder), a controller to process input signals, and a drive circuit to regulate power. The motor receives instructions from the controller, which compares the desired position (input signal) with the actual position (feedback). The controller adjusts the motor's motion to minimize any error, ensuring smooth and accurate operation.



Servo motors operate on the principle of a **closed-loop control system**, where feedback is constantly used to refine performance. This mechanism enables high precision and stability, even under varying loads. Depending on the type, servo motors can perform tasks ranging from limited angle movement to continuous rotation or even converting rotational motion into linear motion.

Basic Operation of Servo Motor

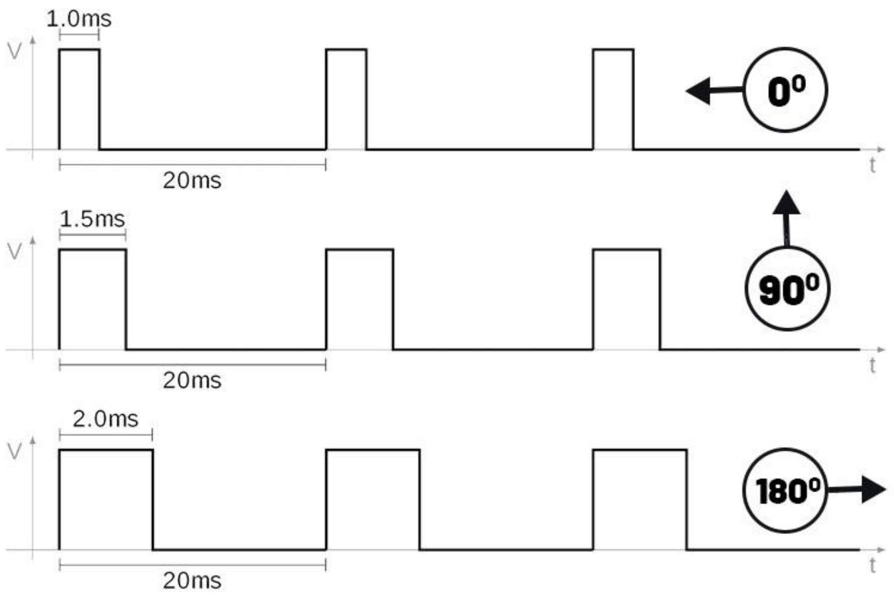
A servo motor operates through a **closed-loop control system** to ensure precise movement and positioning. It includes three key components:

1. **Motor**: Provides rotational or linear movement.
2. **Feedback Mechanism**: Typically a potentiometer or encoder that monitors the motor shaft's position in real-time.
3. **Control Circuit**: Compares the desired position (input signal) with the actual position (feedback) and adjusts the motor's movement to correct any error.

The control system uses this feedback to continuously refine the motor’s position, ensuring it reaches and maintains the desired angle or location.

Servo motors are controlled using **Pulse Width Modulation (PWM)**, a method that involves sending pulses of varying widths to the servo. The width of each pulse determines the position of the motor shaft. Here's how it works:

- **Pulse Frequency**: The servo typically expects a pulse every 20 milliseconds (50 Hz), though many hobby-grade servos accept signals ranging from 40 to 200 Hz.



- **Pulse Width and Position**:
    - A pulse width of **1 ms** corresponds to the minimum position (0°).
    - A pulse width of **1.5 ms** places the motor at the neutral position (90°).
    - A pulse width of **2 ms** moves the motor to the maximum position (180°).

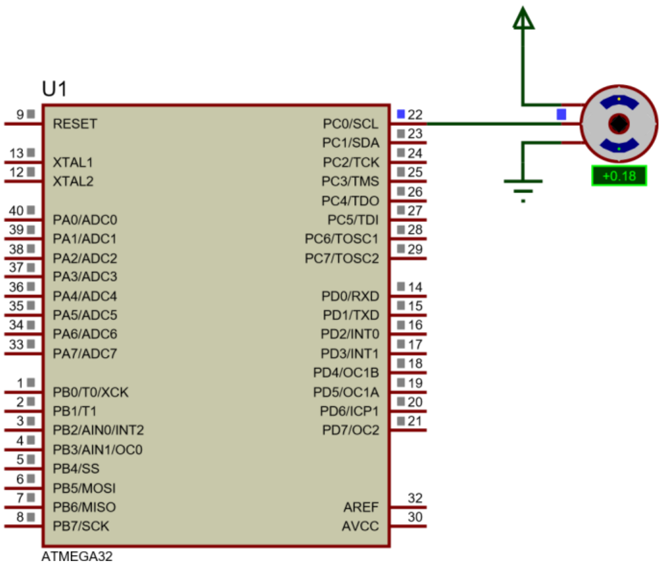
Example : Program to Control a Servo Motor at 0°, 90°, and 180°).

Code:

```
#include <avr/io.h>
#include <util/delay.h>

#define F_CPU 8000000UL // Define the CPU clock speed as 8 MHz

int main(void) {
       DDRC = 0x01;  // Set PC0 as output pin for controlling the servo motor
       PORTC = 0x00;  // Initialize PORTC (PC0) to LOW, so the servo is in its initial 
position

       while(1) {
              // Rotate Motor to 0 degree
              PORTC = 0x01;           // Set PC0 to HIGH (1) to send pulse to servo 
for 0 degree position
              _delay_us(1000);        // Pulse width of 1ms (for 0 degree)
              PORTC = 0x00;           // Set PC0 to LOW (0) to complete the pulse
```

```
            _delay_ms(2000);        // Wait for 2 seconds before the next movement

            // Rotate Motor to 90 degree
            PORTC = 0x01;           // Set PC0 to HIGH (1) to send pulse to servo
for 90 degree position
            _delay_us(1500);        // Pulse width of 1.5ms (for 90 degrees)
            PORTC = 0x00;           // Set PC0 to LOW (0) to complete the pulse
            _delay_ms(2000);        // Wait for 2 seconds before the next movement

            // Rotate Motor to 180 degree
            PORTC = 0x01;           // Set PC0 to HIGH (1) to send pulse to servo
for 180 degree position
            _delay_us(2000);        // Pulse width of 2ms (for 180 degrees)
            PORTC = 0x00;           // Set PC0 to LOW (0) to complete the pulse
            _delay_ms(2000);        // Wait for 2 seconds before repeating the loop
       }
}
```

Circuit Diagram



3 Display Interfacing

Display interfacing in embedded systems serves as the essential link between the system's internal processes and the user. Once sensors have captured data from the environment and actuators have performed the necessary actions, the display provides a means to visualize the system's status, outputs, or feedback. It translates processed signals into human-readable information, enabling users to monitor and interact with the system effectively. For example, a temperature sensor integrated into an embedded system can relay its measurements to a display, showing the current temperature, while an actuator’s operational status, such as the speed of a motor or the position of a valve, can also be communicated visually.

The role of display interfacing becomes particularly important when systems need to provide real-time feedback. Sensors measure environmental parameters such as temperature, humidity, or pressure, and the microcontroller processes this data for meaningful interpretation. The display then shows this processed information, making the system user-friendly and accessible.

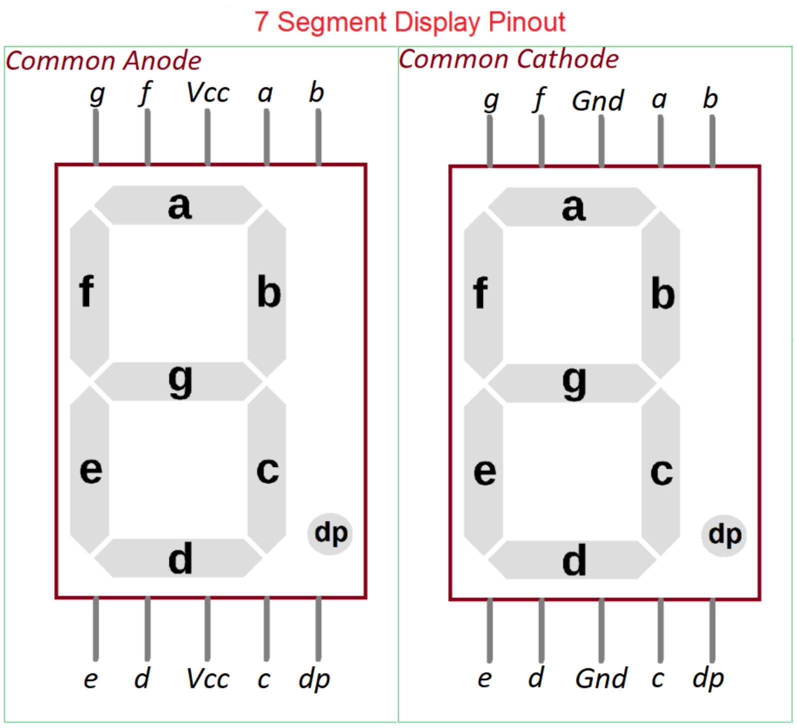
To interface a display, the microcontroller communicates with the display unit through digital signals. Seven-segment displays are commonly used for numeric data representation, especially in applications like digital meters, clocks, and counters. These displays consist of LED segments arranged to form numbers. By activating specific segments, numerical data from sensors or actuators can be displayed clearly. For more complex data, LCDs (Liquid Crystal Displays) are widely used, offering capabilities for alphanumeric and graphical representation. These displays can show detailed information such as sensor readings, system alerts, or even user menus. LCDs often require specific initialization and control commands for effective communication with the microcontroller, making them versatile and essential in advanced embedded systems.

3.1 Seven Segment Display

A seven-segment display is an electronic display device used to represent numbers and limited alphabetic characters. It is widely utilized in embedded systems for applications like digital clocks, calculators, and counters due to its simplicity and cost-effectiveness. The display consists of seven LED segments (labeled ‘a’ to ‘g’) arranged in a rectangular pattern, with an additional dot segment (‘dp’) for decimal points. By selectively illuminating these segments, a wide range of characters can be displayed.

3.1.1 Types of Seven-Segment Displays

Seven-segment displays are classified into two main types based on their electrical configuration:

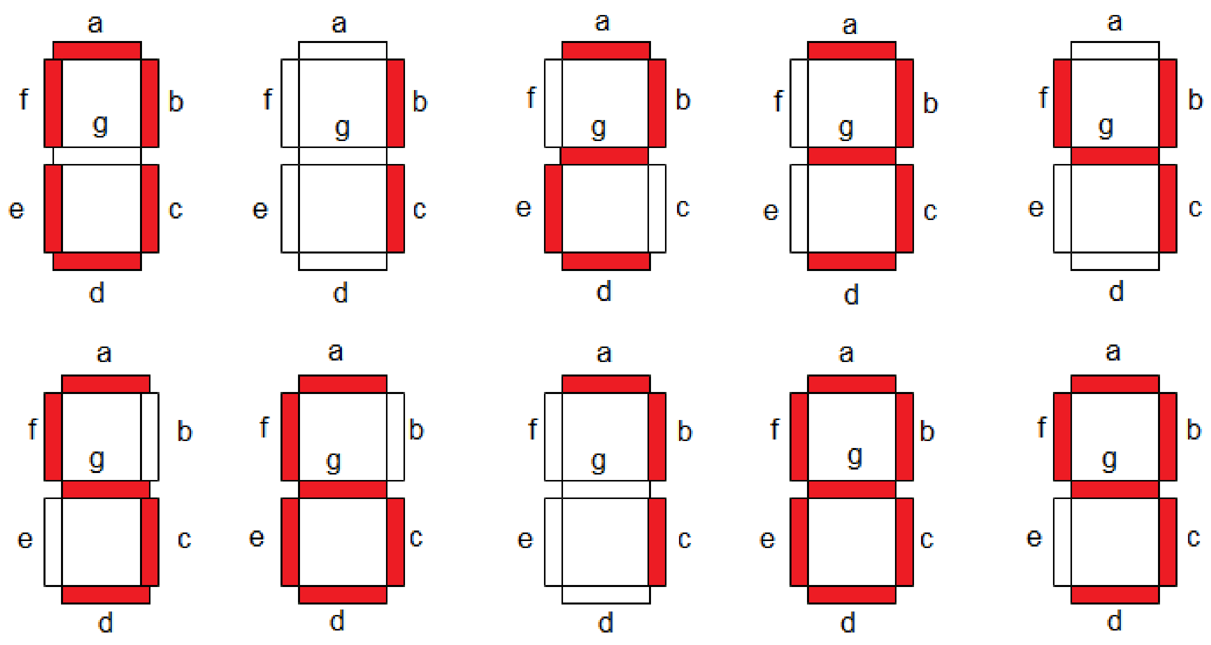


1. **Common Cathode (CC):**In a common cathode display, all the cathode terminals of the LEDs are connected together to a common ground. The individual segments are illuminated by supplying a high (logic 1) voltage to the respective anode pins.

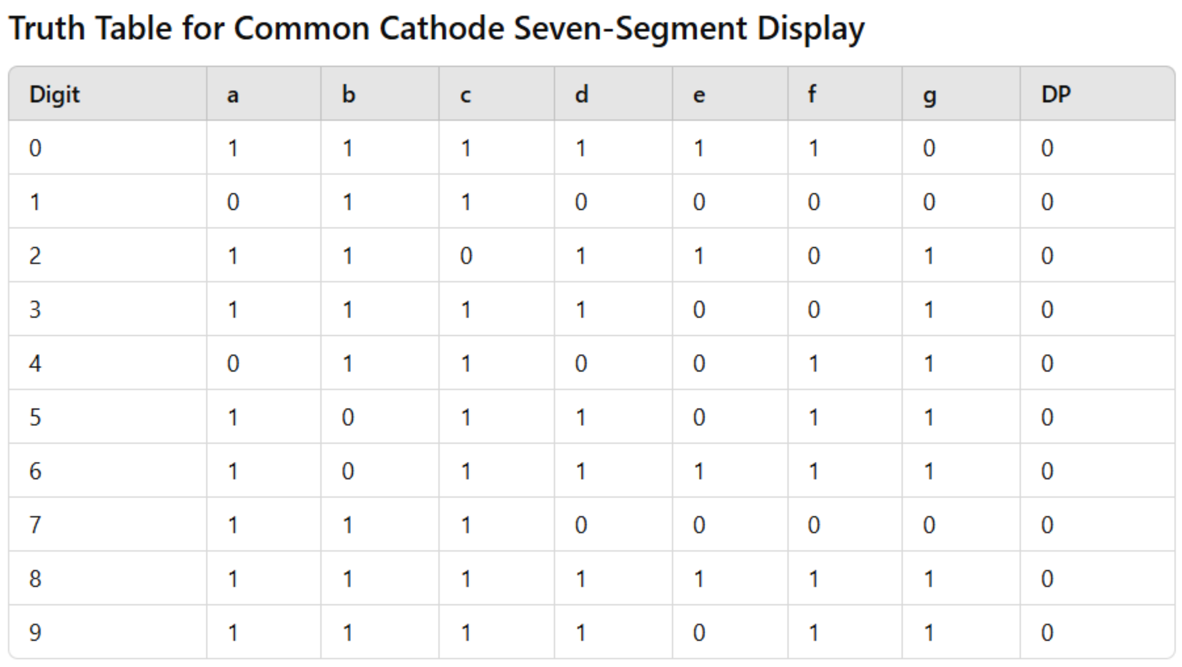
2. **Common Anode (CA):**In a common anode display, all the anode terminals of the LEDs are connected together to a common high voltage. The individual segments are illuminated by grounding (logic 0) the respective cathode pins.

3.1.2 Working of Seven-Segment Displays

A seven-segment display operates by selectively powering its individual segments (labeled ‘a’ to ‘g’) to form numeric digits or specific alphabetic characters. When power is applied to all segments, the display shows the digit **8**, as all segments are illuminated. To display other digits, specific segments are turned OFF. For example, to display the digit **0**, the segment **g** is turned OFF while the rest remain ON.



Seven-segment displays can represent decimal numbers (0–9) and certain alphabetic characters like **A, B, C, D, E, and F**. However, they are limited in their ability to display letters such as **X** or **Z** due to the segment arrangement. Each display unit typically includes a **Decimal Point (DP)**, which can be positioned either on the left or the right of the digit, allowing the display to handle floating-point numbers when necessary. The table below illustrates the segment activation for each digit (0–9). The logic levels (1 or 0) indicate whether a segment is ON or OFF. Note that the logic levels must be reversed for common anode displays.



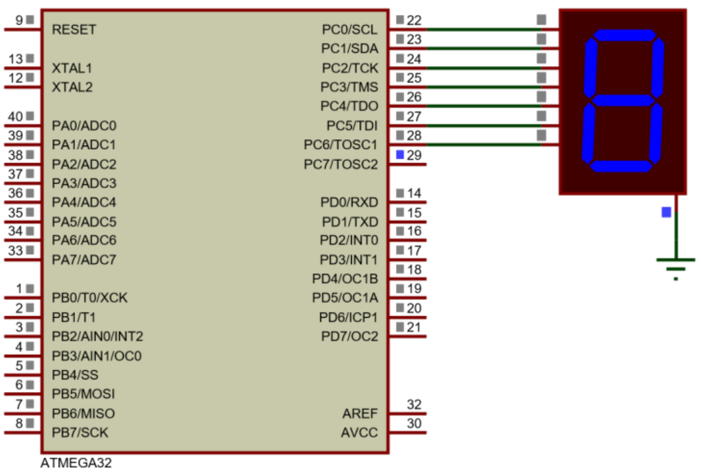
Truth Table for Common Cathode Seven-Segment Display

Digit	a	b	c	d	e	f	g	DP
0	1	1	1	1	1	1	0	0
1	0	1	1	0	0	0	0	0
2	1	1	0	1	1	0	1	0
3	1	1	1	1	0	0	1	0
4	0	1	1	0	0	1	1	0
5	1	0	1	1	0	1	1	0
6	1	0	1	1	1	1	1	0
7	1	1	1	0	0	0	0	0
8	1	1	1	1	1	1	1	0
9	1	1	1	1	0	1	1	0

Code:

```c
/*
 * GccApplication11.c
 *
 * Created: 12/30/2024 5:59:19 PM
 * Author : Deepesh
 */


#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    DDRC |= 0xff;      /* define LED port direction as output */
    PORTC = 0x00;            /* turn off all segments initially */

    /* Hex values for CC display to display digits 0 to 9 */
    char array[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};

    while(1)
    {
        for (int i = 0; i < 10; i++)
        {
            PORTC = array[i]; /* write data to the LED port */
            _delay_ms(1000);     /* wait for 1 second */
        }
    }
       return 0;
}
```

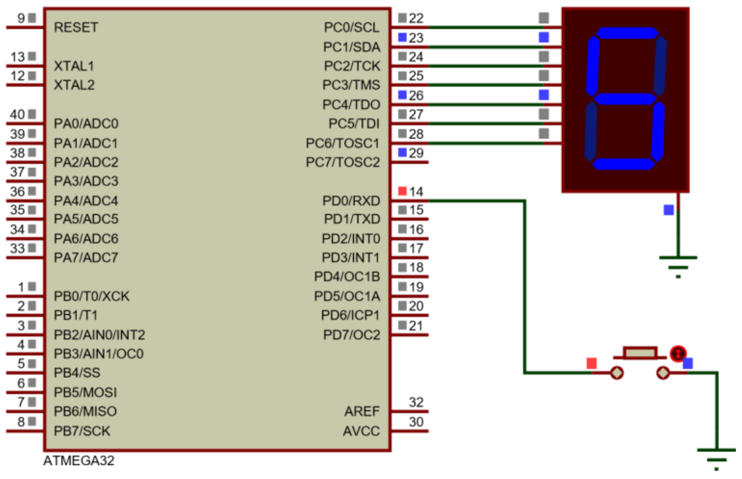
Circuit Diagram:



Code:

```c
/*
 * GccApplication11.c
 *
 * Created: 12/30/2024 5:59:19 PM
 * Author : Deepesh
 */


#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>
#define SWITCH_PIN PIND          /* define switch pin */
#define SWITCH PD0               /* define specific switch pin */

int main(void)
{
    DDRC|= 0xff;     /* define LED port direction as output */
    PORTC  = 0x00;           /* turn off all segments initially */

    DDRD &= ~(1 << SWITCH);     /* define switch pin as input */
    PORTD |= (1 << SWITCH);     /* enable internal pull-up resistor for switch */

    /* Hex values for CC display to display digits 0 to 9 */
    char array[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};
    int count = 0;              /* variable to store count */

    while (1)
    {
        if (!(SWITCH_PIN & (1 << SWITCH))) /* check if switch is pressed */
        {
            _delay_ms(20);      /* debounce delay */
            while (!(SWITCH_PIN & (1 << SWITCH))); /* wait until switch is released */
            _delay_ms(20);      /* debounce delay */

            count++;            /* increment count */
            if (count > 9)     /* reset count after 2 */
                count = 0;

            PORTC = array[count]; /* display count on 7-segment */
        }
    }
}
```

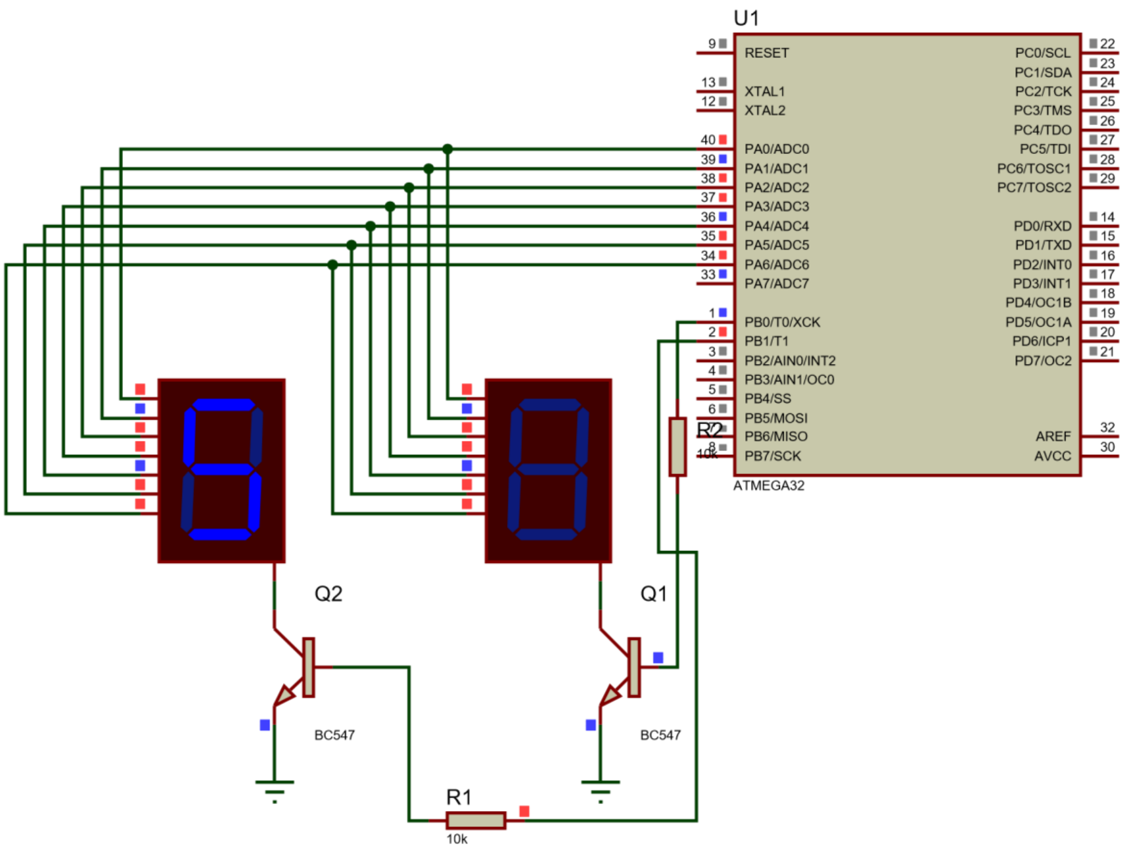
Circuit Diagram:



Code:

```
/*
 * ATmega16_7_Segment_Multiplexed_Display.c
 * Simplified code for counting 0-99 on two 7-segment displays using a single port
 */

#define F_CPU 8000000UL
#include <avr/io.h>
#include <util/delay.h>

#define LED_Direction DDRA      /* Define LED Direction */
#define LED_PORT PORTA          /* Define LED port */
#define Control_Direction DDRB  /* Define Control pin direction */
#define Control_PORT PORTB      /* Define Control port */

/* Hex values for CC display to show digits 0 to 9 */
char array[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};

int main(void)
{
    LED_Direction |= 0xFF;      /* Set LED port as output */
    Control_Direction |= 0x03; /* Set PB0 and PB1 as output for display control */
    Control_PORT &= ~(0x03);   /* Turn off both displays initially */

    int count = 0;             /* Initialize counter */

    while (1)
    {
        for (int i = 0; i < 100; i++) /* Count from 0 to 99 */
        {
            for (int j = 0; j < 2; j++) /* Refresh displays */
            {
                /* Display the ones digit */
                LED_PORT = array[count % 10];
                Control_PORT = (1 << PB0); /* Enable display 1 */
                _delay_ms(10);
                Control_PORT = 0x00;      /* Disable display 1 */

                /* Display the tens digit */
                LED_PORT = array[count / 10];
                Control_PORT = (1 << PB1); /* Enable display 2 */
                _delay_ms(10);
                Control_PORT = 0x00;       /* Disable display 2 */
            }

            count++;
            if (count > 99) count = 0; /* Reset count after 99 */
        }
    }
}
```

Circuit Diagram



3.2 Liquid Crystal Display (LCD)

In embedded systems, Liquid Crystal Displays (LCDs) are widely favored for their versatility and efficiency compared to older technologies like Light Emitting Diodes (LEDs). LCDs are capable of displaying a wide range of information, including numbers, characters, and even complex graphics, making them highly suitable for modern applications. Among the different types, dot-matrix character LCDs are particularly popular in embedded systems due to their simplicity and functionality.

LCDs offer several advantages over LEDs. One of the most significant benefits is their cost-effectiveness, as the declining prices of LCDs have made them accessible for various applications. Their versatility allows them to display alphanumeric characters and graphical data, unlike LEDs, which are generally limited to numbers and a few basic characters. Additionally, LCDs come with a built-in refresh controller that handles display refresh cycles, relieving the CPU of this task. This contrasts with LEDs, which require continuous CPU intervention to maintain the display. Furthermore, LCDs are easier to program for character and graphical displays, simplifying their integration into embedded systems. These advantages make LCDs a preferred choice for a wide array of applications in the field of embedded systems.

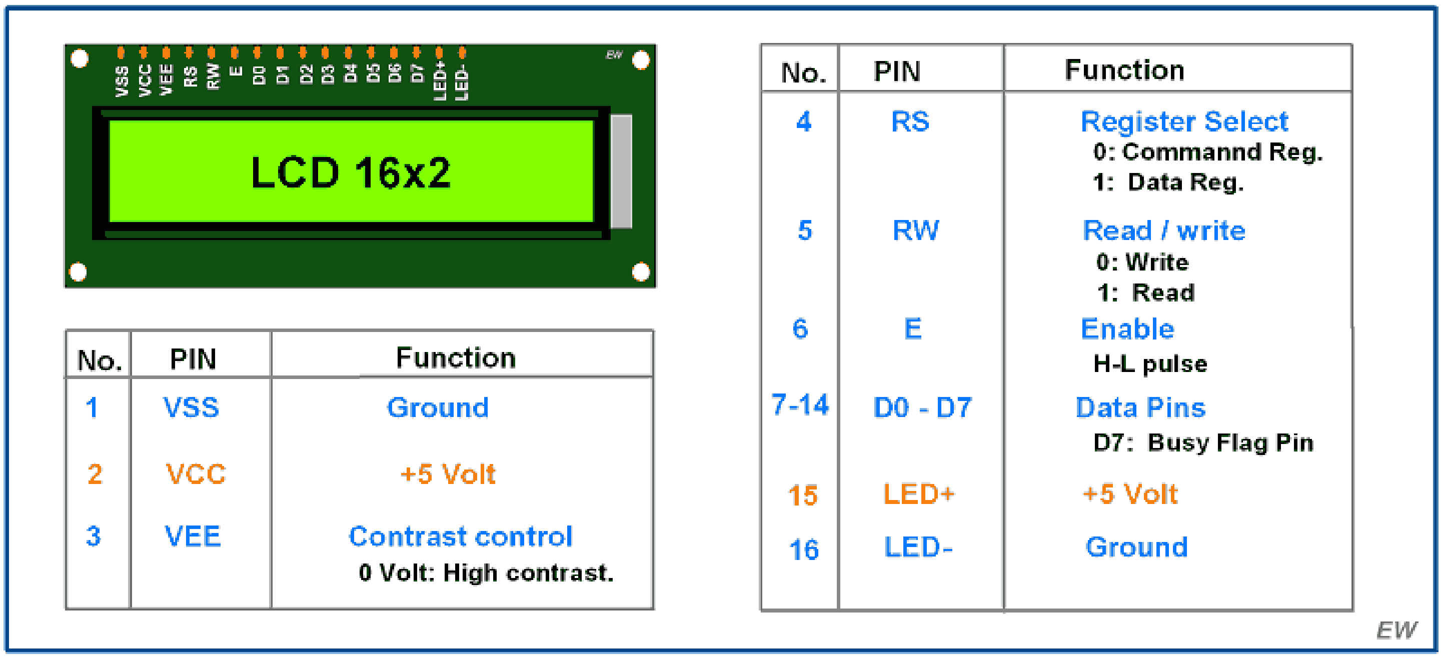
Commonly used LCDs in embedded systems can be categorized based on their display capabilities and configurations. The most popular types include **character LCDs**, **graphic LCDs**, and **segment LCDs**.

- **Character LCDs**: These are typically dot-matrix displays that can show alphanumeric characters in predefined rows and columns, such as 16x2 or 20x4 configurations. They are widely used for displaying simple text and numbers.
- **Graphic LCDs**: These offer greater flexibility as they can display custom images, shapes, and characters, making them suitable for more advanced applications requiring visual graphics.
- **Segment LCDs**: Designed to display specific pre-defined segments (such as digits or symbols), these are common in devices like digital clocks and calculators.

Each type has its specific application, but character LCDs are particularly popular due to their ease of programming and integration into embedded systems.

3.2.1 LCD 16x2 Display: Pin Description and Configuration

The LCD 16x2 display is a widely used alphanumeric display in embedded systems. It features two rows, each capable of displaying 16 characters, with various options for character formatting and backlighting. Here, we provide a clear and beginner-friendly explanation of its pin descriptions, specifications, commands, and configurations.

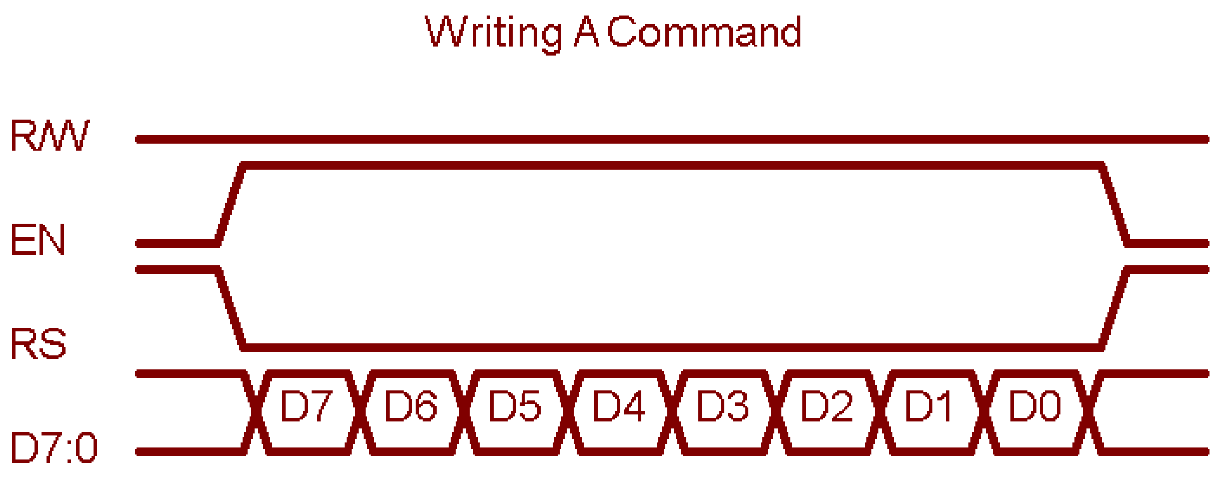


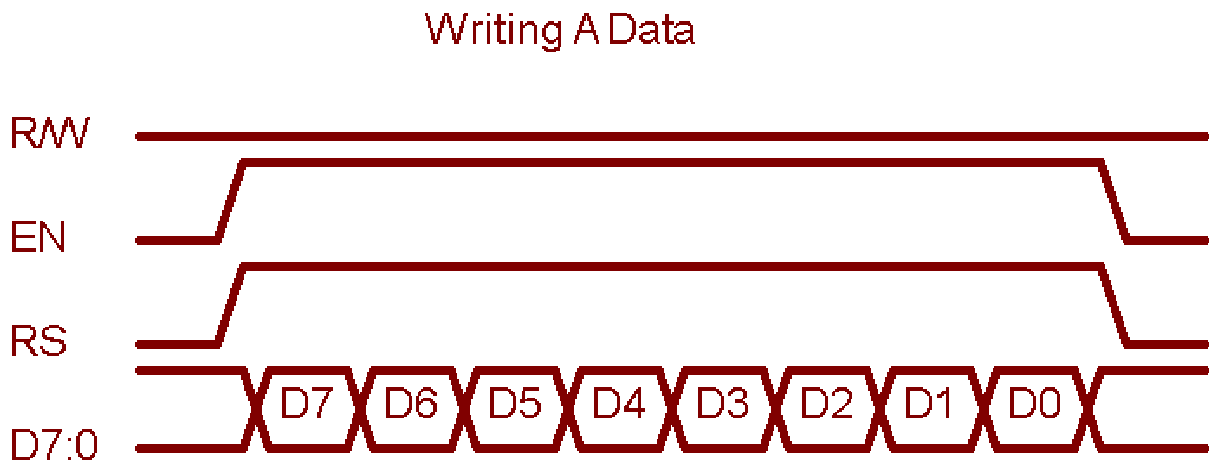
No.	PIN	Function
1	VSS	Ground
2	VCC	+5 Volt
3	VEE	Contrast control 0 Volt: High contrast.

No.	PIN	Function
4	RS	Register Select 0: Commannd Reg. 1: Data Reg.
5	RW	Read / write 0: Write 1: Read
6	E	Enable H-L pulse
7-14	D0 - D7	Data Pins D7: Busy Flag Pin
15	LED+	+5 Volt
16	LED-	Ground

1. **Pins 1 and 2**: These are for power supply connections. Pin 1 is connected to ground (GND), and Pin 2 is connected to a 5V DC supply.
2. **Pin 3 (VEE)**: This pin adjusts the display contrast. By varying the voltage on this pin using a potentiometer (commonly 4.7 kΩ), you can control the contrast. Connecting it to GND provides maximum contrast.
3. **Pin 4 (RS - Register Select)**:
   - RS = 0: The data sent to pins D0-D7 is interpreted as a command.
   - RS = 1: The data sent is treated as content to be displayed.
4. **Pin 5 (RW - Read/Write)**:
   - RW = 0: Data is written to the LCD.
   - RW = 1: Data is read from the LCD.
5. **Pin 6 (E - Enable)**: This pin enables the LCD to latch data on the data pins. A high-to-low pulse (minimum width of 450 ns) is required to register data.

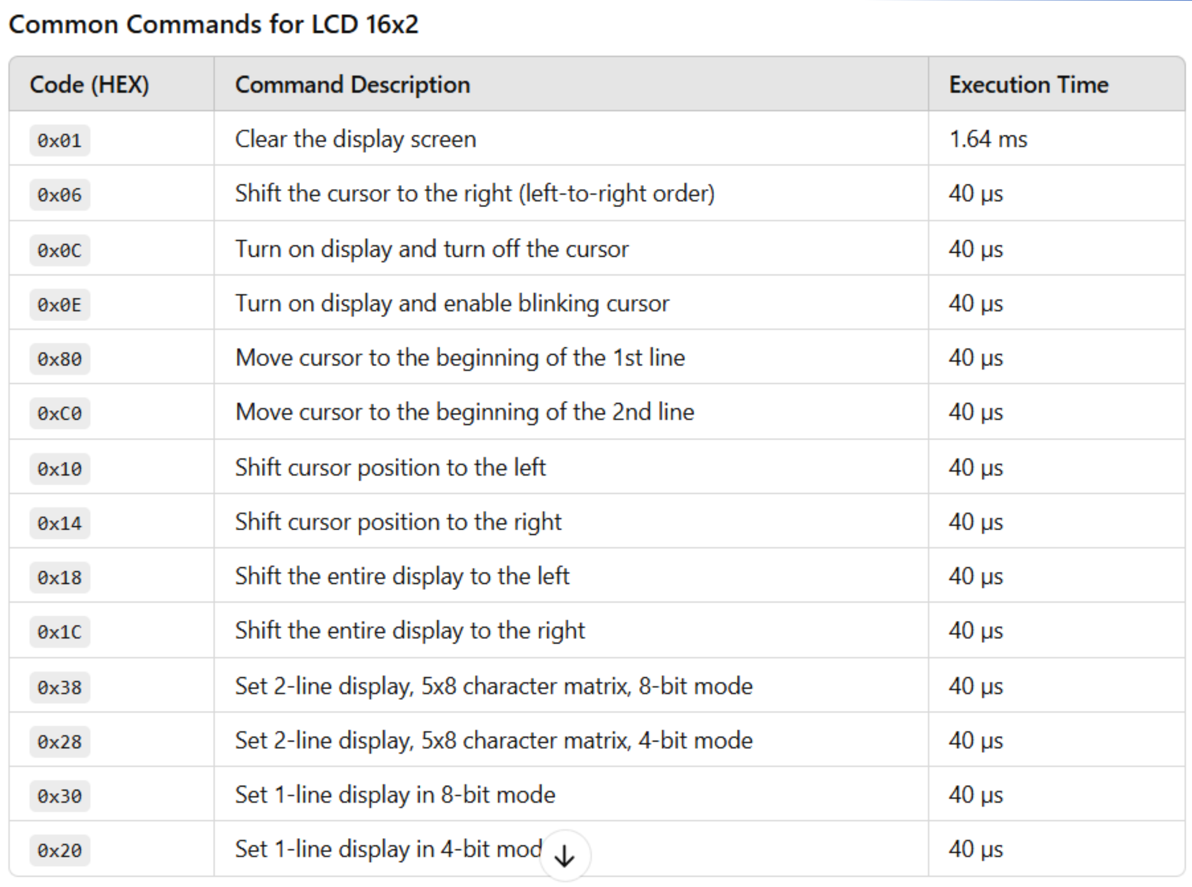
6. **Pins 7 to 14 (D0 to D7)**: These pins serve as the parallel data/command interface. In 8-bit mode, all pins are used. For 4-bit mode, only pins D4-D7 are connected.
7. **Pins 15 and 16 (LED+ and LED-)**: These provide power to the backlight, which improves visibility in low-light conditions.





3.2.2 LCD 16x2 Commands: Overview and Usage

To use an LCD 16x2 display with a microcontroller, it must first be initialized using specific commands. These commands also allow you to control the display, such as clearing it, setting the cursor position, or modifying its appearance. Below is a list of commonly used commands with their functions and execution times.



Common Commands for LCD 16x2

Code (HEX)	Command Description	Execution Time
`0x01`	Clear the display screen	1.64 ms
`0x06`	Shift the cursor to the right (left-to-right order)	40 µs
`0x0C`	Turn on display and turn off the cursor	40 µs
`0x0E`	Turn on display and enable blinking cursor	40 µs
`0x80`	Move cursor to the beginning of the 1st line	40 µs
`0xC0`	Move cursor to the beginning of the 2nd line	40 µs
`0x10`	Shift cursor position to the left	40 µs
`0x14`	Shift cursor position to the right	40 µs
`0x18`	Shift the entire display to the left	40 µs
`0x1C`	Shift the entire display to the right	40 µs
`0x38`	Set 2-line display, 5x8 character matrix, 8-bit mode	40 µs
`0x28`	Set 2-line display, 5x8 character matrix, 4-bit mode	40 µs
`0x30`	Set 1-line display in 8-bit mode	40 µs
`0x20`	Set 1-line display in 4-bit mod	40 µs

To display characters on the LCD, you must send the ASCII code of the character to the module. For instance, to print the letter "H," send the hexadecimal value `0x48` (the ASCII code for "H") to the LCD. The built-in controller of the LCD handles the display process automatically. This command-based interaction makes it straightforward to create a dynamic and user-friendly interface using the LCD 16x2 module.

3.2.3 LCD 16x2 4-bit Mode Configuration

To save GPIO pins on a microcontroller, the LCD 16x2 can operate in 4-bit mode rather than the default 8-bit mode. This setup uses only four of the eight data pins (D4-D7) on the LCD module. Below is a description of the command and how to configure the LCD in 4-bit mode. When configuring the LCD 16x2 in 4-bit mode, the following steps must be followed. The command structure consists of several bits, with each bit representing a specific function:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	DL	N	F	-	-

- **D1 and D0**: These bits should be set to 0.
- **D7 and D6**: These bits should be set to 0.
- **D5**: This bit should be set to 1.
- **D4**: 1 (Set to 1 to enable 4-bit mode)
- **D3 to D0**: Set to 0 (No special function)
- **DL (Interface Data Length)**:
    - `0` indicates 4-bit mode
    - `1` indicates 8-bit mode
- **N (Number of Display Lines)**:
    - `0` for 1 line
    - `1` for 2 lines
- **F (Character Font)**:
    - `0` for 5x8 dots (standard character format)
    - `1` for 5x10 dots (extended character format)

To switch the LCD from 8-bit to 4-bit mode, send a command to the LCD where the higher nibble (D4-D7) is set to `0x02`. This initializes the module in 4-bit mode. Following this command, any subsequent command can use a lower nibble as `2` (e.g., `0x32`, `0x42`, etc.), except `0x22`, which is reserved and used for 8-bit mode. This configuration significantly reduces the number of GPIO pins needed from 8 to 4, making it more convenient for use in embedded systems with limited pin availability.

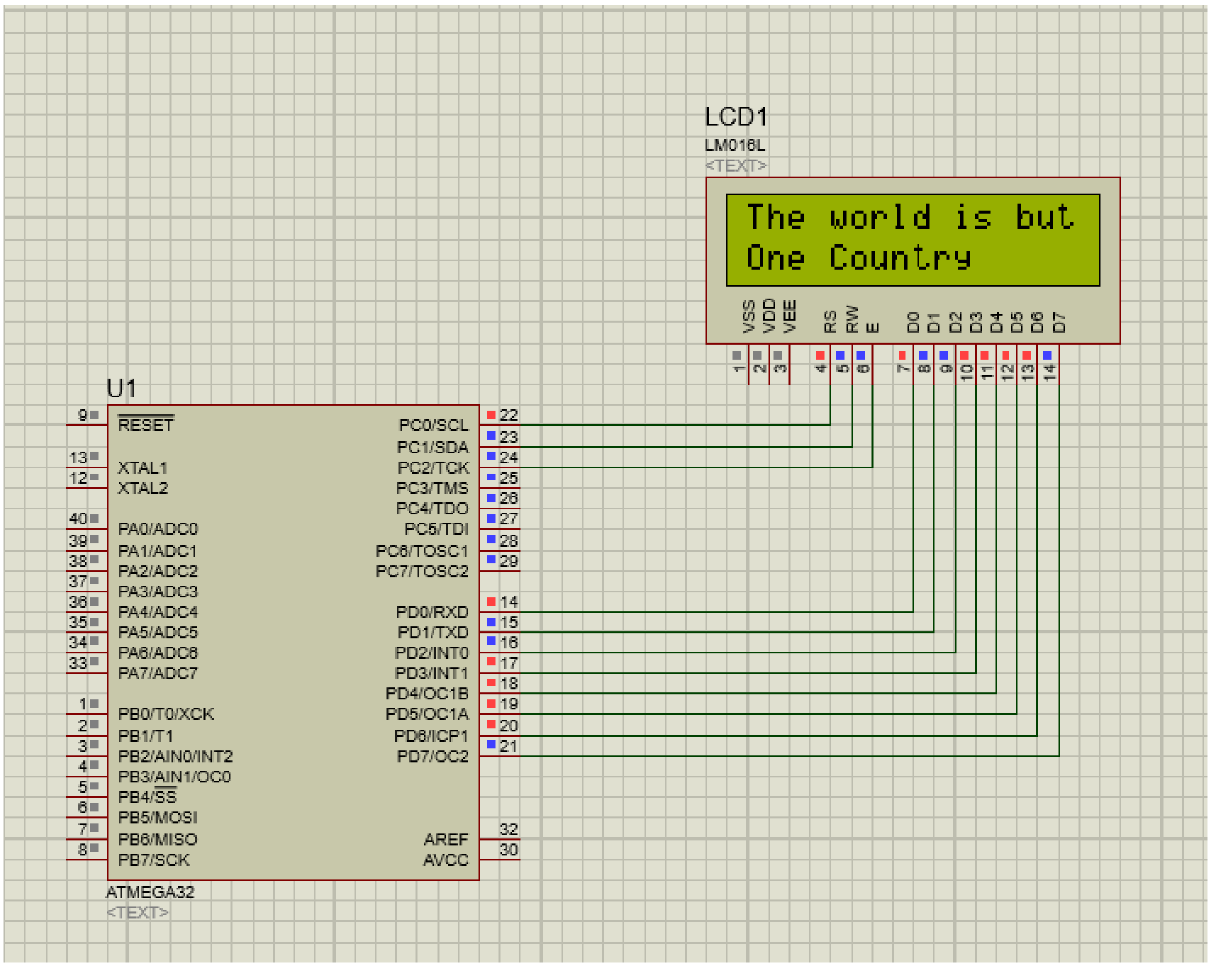
Code:

```
/*
 * GccApplication12.c
 *
 * Created: 12/30/2024 8:48:21 PM
 * Author : Deepesh
 */
```

```
#define F_CPU 8000000UL                      /* Define CPU Frequency e.g. here 8MHz */
#include <avr/io.h>                   /* Include AVR std. library file */
#include <util/delay.h>                      /* Include inbuilt defined Delay header file
*/

#define LCD_Data_Dir DDRD             /* Define LCD data port direction */
#define LCD_Command_Dir DDRC                 /* Define LCD command port direction register
*/
#define LCD_Data_Port PORTD           /* Define LCD data port */
#define LCD_Command_Port PORTC               /* Define LCD data port */
#define RS PC0                               /* Define Register Select (data/command
reg.)pin */
#define RW PC1                               /* Define Read/Write signal pin */
#define EN PC2                               /* Define Enable signal pin */


void LCD_Command(unsigned char cmnd)
{
       LCD_Data_Port= cmnd;
       LCD_Command_Port &= ~(1<<RS);         /* RS=0 command reg. */
       LCD_Command_Port &= ~(1<<RW);         /* RW=0 Write operation */
       LCD_Command_Port |= (1<<EN);          /* Enable pulse */
       _delay_us(1);
       LCD_Command_Port &= ~(1<<EN);
       _delay_ms(3);
}

void LCD_Char (unsigned char char_data)      /* LCD data write function */
{
       LCD_Data_Port= char_data;
       LCD_Command_Port |= (1<<RS);          /* RS=1 Data reg. */
       LCD_Command_Port &= ~(1<<RW);         /* RW=0 write operation */
       LCD_Command_Port |= (1<<EN);          /* Enable Pulse */
       _delay_us(1);
       LCD_Command_Port &= ~(1<<EN);
       _delay_ms(1);
}

void LCD_Init (void)                  /* LCD Initialize function */
{
       LCD_Command_Dir = 0xFF;               /* Make LCD command port direction as o/p */
       LCD_Data_Dir = 0xFF;           /* Make LCD data port direction as o/p */
       _delay_ms(20);                        /* LCD Power ON delay always >15ms */

       LCD_Command (0x38);            /* Initialization of 16X2 LCD in 8bit mode */
       LCD_Command (0x0C);            /* Display ON Cursor OFF */
       LCD_Command (0x06);            /* Auto Increment cursor */
       LCD_Command (0x01);            /* Clear display */
       LCD_Command (0x80);            /* Cursor at home position */
}

void LCD_String (char *str)           /* Send string to LCD function */
{
       int i;
       for(i=0;str[i]!=0;i++)                /* Send each char of string till the NULL */
       {
              LCD_Char (str[i]);
       }
}

void LCD_String_xy (char row, char pos, char *str)/* Send string to LCD with xy
position */
```

```
{
	if (row == 0 && pos<16)
	LCD_Command((pos & 0x0F)|0x80);    /* Command of first row and required 
position<16 */
	else if (row == 1 && pos<16)
	LCD_Command((pos & 0x0F)|0xC0);    /* Command of first row and required 
position<16 */
	LCD_String(str);            /* Call LCD string function */
}

void LCD_Clear()
{
	LCD_Command (0x01);         /* clear display */
	LCD_Command (0x80);         /* cursor at home position */
}

int main()
{

	LCD_Init();                 /* Initialize LCD */

	LCD_String("The world is but ");   /* write string on 1st line of LCD*/
	LCD_Command(0xC0);          /* Go to 2nd line*/
	LCD_String("One Country");  /* Write string on 2nd line*/

	return 0;
}
```

Circuit Diagram


LCD1
LM016L
<TEXT>
The world is but
One Country
VSS
VDD
VEE
RS
RW
E
D0
D1
D2
D3
D4
D5
D6
D7
U1
RESET
XTAL1
XTAL2
PA0/ADC0
PA1/ADC1
PA2/ADC2
PA3/ADC3
PA4/ADC4
PA5/ADC5
PA6/ADC6
PA7/ADC7
PB0/T0/XCK
PB1/T1
PB2/AIN0/INT2
PB3/AIN1/OC0
PB4/SS
PB5/MOSI
PB6/MISO
PB7/SCK
PC0/SCL
PC1/SDA
PC2/TCK
PC3/TMS
PC4/TDO
PC5/TDI
PC6/TOSC1
PC7/TOSC2
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/OC1B
PD5/OC1A
PD6/ICP1
PD7/OC2
AREF
AVCC
ATMEGA32
<TEXT>