Department of Physics
University of Colombo

PH3032 – Embedded System Laboratory (2024)

Chemical Test Tube Mixer

A.M.I.H.K.Abeysinghe

Reg No: 2021s18738

Index: s16083

• Abstraction

In laboratory settings, precise and consistent mixing of chemical solutions is essential for accurate experimental results. This project aims to develop an automated chemical test tube mixer using the ATmega328P microcontroller and a stepper motor to ensure uniform mixing across various solutions. The mixer offers adjustable rotation angles (180°, 360°, and 30°) and speeds, controlled via a potentiometer, with an LCD displaying real-time motor speed in rotations per minute (RPM). The system was built using components including the ATmega328P, stepper motor, A4988 driver, and an LCD display, with user interaction handled through buttons. The motor speed and rotation were controlled using precise pulses and adjustable delays. Initial tests confirmed that the system performed as expected, providing reliable and accurate mixing. However, extended operation led to performance issues, such as delays in motor response, caused by the overheating of the voltage regulator. The automated mixer successfully improved consistency and precision in chemical mixing, making it a valuable tool for laboratory experiments. The project highlights the importance of effective power regulation in maintaining system stability during continuous use, suggesting potential improvements like enhanced cooling mechanisms and switching regulators to optimize performance over time.

• Introduction

In laboratory settings, the precise and uniform mixing of chemical solutions is crucial for ensuring accurate reactions and results. Manual mixing methods, while commonly used, can be time-consuming and may introduce inconsistencies. Automating the mixing process with microcontrollers and stepper motors provides a more reliable, efficient, and controlled approach. Stepper motors, known for their precision in rotational movement, are particularly well-suited for applications requiring specific angle control.

This project focuses on the development of an automated chemical test tube mixer, powered by the ATmega328P microcontroller and a stepper motor. Designed for laboratory use, the mixer offers customizable speeds and three distinct mixing types: 180-degree, 30-degree, and 360-degree rotations. User-friendly features include on/off control, speed adjustment, mixing type selection, and the ability to display all relevant details on an LCD screen for ease of monitoring and control.

Automating the mixing process significantly improves both accuracy and consistency, reducing the likelihood of human error. This makes the device highly valuable in experiments requiring precision, particularly in fields such as chemistry and biology where uniform mixing is essential for reliable results. The ability to select different mixing angles—180-degree, 30-degree, and 360-degree rotations—adds flexibility, allowing the mixer to accommodate a variety of experimental needs and processes. This versatility, combined with enhanced control over mixing speed and type, ensures that the device can be adapted to a range of applications in the laboratory setting.

The hypothesis is that the automated test tube mixer will provide consistent and precise mixing across different solutions, offering greater control than traditional manual methods. The adjustable speed and rotational angles are expected to improve the reliability and repeatability of the mixing process.

• Methodology

This project aimed to develop an automated chemical test tube mixer with adjustable rotation angles and speeds, using the ATmega328P microcontroller. The system is designed to rotate test tubes in 180°, 360°, and 30° increments, with control over speed through a potentiometer.

➢ Components

1. ATMega328p Micro-controller
2. Stepper Motor
3. Stepper Motor Controller (A4988 driver)
4. LCD Display
5. I2C Module
6. Voltage Regulator
7. Push button
8. On/Off button
9. Potential Meter

The components were wired to the ATmega328P as per the pin configurations in the code. The stepper motor was connected to the A4988 driver, and the driver was connected to the ATmega328P, with the `DIR_PIN` (PD0) controlling the motor direction and `STEP_PIN` (PD1) sending step pulses. Buttons for selecting the rotation modes were connected to pins PD2 (INT0), PD3 (INT1), and PD4 (PCINT20).

➢ Button and Interrupt Initialization

The three buttons, each corresponding to a specific motor rotation angle (180°, 360°, and 30°), were initialized using **external interrupts** and **pin change interrupts** on the ATmega328P. The system uses interrupts to ensure responsive handling of button presses, minimizing the delay in starting motor rotations once a button is pressed.

Each button is connected to a specific pin on Port D, which is configured as an input with a pull-up resistor enabled. Interrupts are triggered by falling edges (when the button is pressed). The following code segment initializes the buttons and their respective interrupts:

```
 void buttons_init() {
    DDRD &= ~(1 << PD2) & ~(1 << PD3);  // Set PD2 and PD3 as input (INT0, INT1)
    PORTD |= (1 << PD2) | (1 << PD3);   // Enable pull-up resistors for PD2, PD3
    EICRA |= (1 << ISC01) | (1 << ISC11); // Set for falling edge trigger on
INT0, INT1
    EIMSK |= (1 << INT0) | (1 << INT1);   // Enable INT0 and INT1 interrupts

    // PCINT20 (PD4) for 30 degree rotation
    DDRD &= ~(1 << PD4);  // Set PD4 as input
    PORTD |= (1 << PD4);  // Enable pull-up resistor for PD4
    PCICR |= (1 << PCIE2); // Enable pin change interrupt for PCINT20
    PCMSK2 |= (1 << PCINT20);  // Enable interrupt for PD4 (PCINT20)
}
```

Button 1 (180° rotation) is connected to PD2, and an interrupt service routine (ISR) is triggered by a falling edge on INT0. **Button 2 (360° rotation)** is connected to PD3, and an ISR is triggered by a falling edge on INT1. **Button 3 (30° rotation)** is connected to PD4, and pin change interrupts are used to detect a button press.

The buttons are configured in the `buttons_init()` function, which enables the respective interrupts. This ensures that pressing any of the buttons triggers the corresponding ISR, setting flags to indicate which rotation mode is active.

➢ Stepper Motor Control

The stepper motor is controlled by sending pulses to the A4988 driver via the `STEP_PIN`. Each pulse moves the motor by 1.8°, corresponding to one step. The rotation direction is determined by the state of the `DIR_PIN`.

- **180° rotation** requires 100 steps (180° / 1.8° per step).
- **360° rotation** requires 200 steps (360° / 1.8° per step).
- **30° rotation** requires 17 steps (30° / 1.8° per step, rounded).

Two functions, `rotate_forward()` and `rotate_backward()`, are used to move the motor forward and backward by the required number of steps:

```
void rotate_forward(int steps) {
    for (int i = 0; i < steps; i++) {
        PORTD |= (1 << STEP_PIN);  // Pulse HIGH
        _delay_us(currentDelay);   // Delay based on speed control
        PORTD &= ~(1 << STEP_PIN); // Pulse LOW
        _delay_us(currentDelay);
    }
}

void rotate_backward(int steps) {
    for (int i = 0; i < steps; i++) {
        PORTD |= (1 << STEP_PIN);  // Pulse HIGH
        _delay_us(currentDelay);
        PORTD &= ~(1 << STEP_PIN); // Pulse LOW
        _delay_us(currentDelay);
    }
}
```

Pulse generation: The motor steps are driven by toggling the **STEP_PIN** from HIGH to LOW. Each pulse advances the motor by 1.8°. By calling `rotate_forward()` with a specific number of steps, the motor can rotate by the desired angle.

Delays: The `currentDelay` value controls the time between each pulse, thus adjusting the motor speed.

To ensure the motor turns in the correct direction, the **DIR_PIN** is set before calling the rotate function: For clockwise rotation, the **DIR_PIN** is set to HIGH. For counterclockwise rotation, the **DIR_PIN** is set to LOW.This allows the system to rotate the motor in both directions when necessary (for instance, rotating forward and then returning to the starting position).

➢ Speed Control Using ADC

The motor speed is controlled by the potentiometer, which is connected to the ADC (Analog-to-Digital Converter) of the ATmega328P. The potentiometer provides a variable voltage that the ADC reads and converts into a digital value between 0 and 1023. This value is used to determine the delay between motor steps, effectively controlling the rotation speed.

The ADC is initialized and the potentiometer value is read as shown below:

```
void adc_init() {
    ADMUX = (1 << REFS0);  // Use AVcc as reference voltage
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1); // Enable ADC, set
prescaler to 64
}

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 for conversion to complete
    return ADC;
}
```

The speed of the motor is controlled by adjusting the delay between step pulses, which is determined by reading the potentiometer value through the ADC. The `adc_init()` function is set up with a reference voltage of **AVcc** (typically 5V) and a prescaler of 64, which adjusts the ADC clock frequency for proper operation and the `adc_read()` function reads the potentiometer value from **ADC0**. The potentiometer reading directly affects the `currentDelay` variable, which determines how fast the pulses are sent to the stepper motor. A lower `currentDelay` value results in faster motor rotation, while a higher value slows it down.

The potentiometer output is mapped to a delay value that adjusts the motor speed: A higher ADC value results in a shorter delay between pulses, increasing the motor speed. A lower ADC value increases the delay, reducing the motor speed.

➢ Displaying Speed on LCD

The motor speed is calculated based on the step delay and displayed on the LCD. The system uses the I2C communication protocol to send data to the LCD, with the following code calculating and displaying the speed in rotations per minute (RPM):

```
int calculate_rpm(uint16_t delay) {
    float step_freq = 1.0 / (2.0 * delay * 1e-6);  // Step frequency in Hz
    int rpm = (step_freq * 60.0) / 200.0;  // 200 steps per revolution
    return rpm;
}
```

The step frequency is calculated based on the `currentDelay` between step pulses. The `currentDelay` is in microseconds, so this calculation converts it to a frequency in hertz. Since the motor requires 200 steps to complete one full revolution (360°), the RPM is calculated by multiplying the step frequency by 60 and dividing by 200.

The calculated RPM is displayed on the LCD:

```
void display_speed(int rpm) {
    lcd_clear();
    lcd_set_cursor(0, 0);
    lcd_printf("Speed: %d RPM", rpm);
}
```

This function clears the display, sets the cursor to the top-left position, and prints the current speed in RPM. The LCD updates dynamically as the user adjusts the potentiometer, providing real-time feedback on motor speed.

➢ Operation of the System

After initialization, the system continuously monitors the button presses to determine the rotation mode. The function `main()` contains the logic that checks for the selected rotation mode and calls the appropriate rotation function. For example, if the 180° rotation mode is selected, the system performs the following sequence:

```
if (rotationFlag == 1) {
    rotate_forward(100);  // Rotate forward 100 steps (180°)
    rotate_backward(100); // Return to the starting position
    rotationFlag = 0;  // Reset the flag
}
```

The **rotationFlag** is set by the corresponding interrupt service routine (ISR) when a button is pressed, and once the motor completes its rotation, the flag is reset. The motor speed is dynamically adjusted based on the potentiometer input, and the calculated RPM is displayed on the LCD throughout operation.

➢ Testing and Verification

The system was tested by pressing each button and observing the corresponding motor rotation. The motor successfully performed rotations of 180°, 360°, and 30° as expected. The speed control through the potentiometer was verified by adjusting the knob and observing the change in rotation speed, as displayed on the LCD.

The system was also tested for consistency and accuracy. Each rotation mode worked correctly, and the motor returned to its starting position after completing each rotation. Speed readings on the LCD were accurate and responsive to potentiometer adjustments.

- ## Results and Analysis

1. System Behavior

Upon initial testing, the stepper motor control system operated as expected, accurately performing rotations and responding to user inputs. In the early phases of operation, the motor:

Rotated to the specified angles (180°, 360°, and 30°) based on button inputs, with each button press corresponding to the correct number of steps (100, 200, and 17 steps, respectively). Speed control using the potentiometer worked efficiently, allowing the motor to adjust its rotation speed based on the user input. The LCD display correctly showed the RPM values in real time.

However, after the device was left powered on for an extended period, performance issues began to emerge.

2. Observed Delays and Performance Degradation

After operating continuously for a while, the system displayed noticeable delays in motor response and reduced accuracy in executing rotational commands. In particular:

Button Press Delays: At times, there was a noticeable lag between pressing a button and the motor responding. In some cases, the motor failed to execute the commanded rotations entirely. **Inconsistent Speed Control**: The motor speed did not consistently follow changes in the potentiometer setting, with the displayed RPM values becoming unstable after prolonged use.

These performance issues were not present in the early stages of testing but became prominent during extended operation.

3. Possible Cause: Overheating of Voltage Regulator

Upon further investigation, it was noted that the device utilized a voltage regulator to convert 12V to both 5V and 12V supplies for different parts of the system. The voltage regulator, operating over an extended period, likely contributed to the issues due to overheating.

Voltage Regulator Heating: The voltage regulator is responsible for stepping down the input voltage to the necessary levels (12V for the motor and 5V for the control logic). During continuous operation, the regulator can become excessively hot, potentially reducing its ability to maintain a stable output voltage. This can result in:

Voltage Instability: As the voltage regulator heats up, it may fail to provide consistent voltage levels, leading to fluctuations in the power supplied to the microcontroller and motor driver. This instability can directly affect the performance of the motor control system, causing delayed or inaccurate motor rotations.

Circuit Protection Activation: In some cases, the voltage regulator might activate thermal protection mechanisms, temporarily reducing or cutting off power to prevent damage. This could explain the sporadic delays and failure of the motor to respond.

4. Effect on Motor Operation

The overheating issue directly impacted the motor ability to rotate at the expected speeds and angles:

Delayed Motor Response: Voltage instability caused by overheating can interfere with the stepper motor driver ability to deliver accurate pulse signals to the motor, resulting in delayed or incomplete rotations.

Fluctuating RPM: As the power supply to the motor became unstable, the calculated RPM values displayed on the LCD became inconsistent, and the actual motor speed no longer matched the user input from the potentiometer.

5. Potential Solutions to Overheating

To mitigate the impact of overheating on the system performance, the following solutions could be considered:

Improved Heat Dissipation: Adding a heat sink to the voltage regulator or using a more efficient cooling mechanism (such as a small fan) could help dissipate heat and maintain stable voltage levels.

Switching to a Switching Regulator: Replacing the linear voltage regulator with a switching regulator (buck converter) could improve efficiency, as switching regulators produce less heat and are better suited for continuous operation under load.

Power Supply Separation: Using separate power supplies for the motor (12V) and control circuitry (5V) could reduce the load on the voltage regulator and help maintain stability.

6. Theoretical Analysis of System Performance

In the early stages of operation, the system performed as expected:

Motor Rotations: Button presses resulted in accurate motor rotations based on the code calculation of the required steps for each angle (100 steps for 180°, 200 steps for 360°, and 17 steps for 30°).

Speed Control: The potentiometer-controlled speed was mapped to an appropriate delay between steps, which directly influenced the motor RPM. This was verified by the formula

```
RPM = (60 × 1000000) / (step_delay × steps_per_revolution)
```

where changes in the potentiometer value were reflected in both motor speed and the displayed RPM values. However, as the device heated up, the overheating issue introduced inconsistencies in the performance:

Inconsistent Speed and Delay: The motor actual speed no longer matched the displayed RPM, and delays were observed in motor response.

• References

- ❖ Holton, W. C., & Sze, S. (1998, July 20). *Semiconductor device | Electronics, Physics, & Applications*. Encyclopedia Britannica. https://www.britannica.com/technology/semiconductor-device
- ❖ Wikipedia contributors. (2024, September 10). *Voltage regulator*. Wikipedia. https://en.wikipedia.org/wiki/Voltage_regulator
- ❖ *Understanding How a Voltage Regulator Works*. (2021, July 6). Analog Devices. https://www.analog.com/en/resources/technical-articles/how-voltage-regulator-works.html
- ❖ Libretexts. (2024, April 30). *7.2: Phase Space Visualization*. Mathematics LibreTexts. https://math.libretexts.org/Bookshelves/Scientific_Computing_Simulations_and_Modeling/Introduction_to_the_Modeling_and_Analysis_of_Complex_Systems_(Sayama)/07%3A_ContinuousTime_Models_II__Analysis/7.02%3A_Phase_Space_Visualization
- ❖ *Stepper Motor Basics*. (n.d.). Oriental Motor U.S.A. Corp. https://www.orientalmotor.com/stepper-motors/technology/stepper-motor-basics.html
- ❖ *Arduino and Stepper Motor Configuration*. (n.d.). https://docs.arduino.cc/learn/electronics/stepper-motors/
- ❖ *Pololu - Stepper Motor Drivers*. (n.d.). https://www.pololu.com/category/120/stepper-motor-drivers

• APPENDIX

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

#define STEP_PIN PB2      // Pin connected to STEP on A4988
#define DIR_PIN PB1       // Pin connected to DIR on A4988
#define BUTTON_180_PIN PD2 // Button 1 (180° rotation)
#define BUTTON_360_PIN PD3 // Button 2 (360° rotation)
#define BUTTON_30_PIN PD4  // Button 3 (30° rotation)

int f180 = 0;
int f360 = 0;
int f30 = 0;

#define STEPS_PER_REV 200  // Number of steps per full revolution for your motor
(1.8° per step)

// Function to initialize the buttons
```

```
void buttons_init(void) {
    DDRD &= ~((1 << BUTTON_180_PIN) | (1 << BUTTON_360_PIN) | (1 <<
BUTTON_30_PIN));  // Set buttons as inputs
    PORTD |= (1 << BUTTON_180_PIN) | (1 << BUTTON_360_PIN) | (1 <<
BUTTON_30_PIN);    // Enable pull-up resistors
}

// Function to read the state of button 1 (180°)
uint8_t button_180_pressed(void) {
    return !(PIND & (1 << BUTTON_180_PIN));  // Button press is active-low
}

// Function to read the state of button 2 (360°)
uint8_t button_360_pressed(void) {
    return !(PIND & (1 << BUTTON_360_PIN));  // Button press is active-low
}

// Function to read the state of button 3 (30°)
uint8_t button_30_pressed(void) {
    return !(PIND & (1 << BUTTON_30_PIN));  // Button press is active-low
}

// Function to initialize the ADC (for speed control)
void adc_init(void) {
    ADMUX = (1 << REFS0);  // Set reference voltage to AVcc, and select ADC0 (PC0
pin)
    ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1);  // Enable ADC, set
prescaler to 64
}

// Function to read the ADC value from ADC0 (PC0)
uint16_t adc_read(void) {
    ADCSRA |= (1 << ADSC);  // Start conversion
    while (ADCSRA & (1 << ADSC));  // Wait for conversion to complete
    return ADC;  // Return the ADC value (10-bit)
}

// Function to provide a variable microsecond delay (loop-based)
void variable_delay_us(uint16_t delay) {
    while (delay--) {
        _delay_us(1);
    }
}

// Function to pulse the step pin
```

```c
void stepper_step(uint16_t delay) {
    PORTB |= (1 << STEP_PIN);  // Set STEP_PIN high
    _delay_us(1);              // Short delay to ensure the pulse is registered
    PORTB &= ~(1 << STEP_PIN); // Set STEP_PIN low
    variable_delay_us(delay);  // Delay between steps
}

// Rotate stepper motor forward a number of steps
void rotate_forward(uint16_t steps, uint16_t delay) {
    PORTB |= (1 << DIR_PIN);  // Set direction forward
    for (uint16_t i = 0; i < steps; i++) {
        stepper_step(delay);
    }
}

int main(void) {
    // Configure STEP_PIN and DIR_PIN as outputs
    DDRB |= (1 << STEP_PIN) | (1 << DIR_PIN);

    // Initialize buttons and ADC
    buttons_init();
    adc_init();

    uint16_t adc_value;
    uint16_t step_delay;
    uint16_t steps;

    while (1) {
        // Read potentiometer value for speed control
        adc_value = adc_read();
        step_delay = 1023 - adc_value;  // Map ADC value to delay (inverted for
higher speed)

        // Check if any button is pressed and set the mode accordingly
        if (button_180_pressed()) {
            f180 = 1;
            f360 = f30 = 0;  // Reset other modes
            steps = 100;     // 180° rotation
            _delay_ms(200);  // Debounce delay
        } else if (button_360_pressed()) {
            f360 = 1;
            f180 = f30 = 0;  // Reset other modes
            steps = 200;     // 360° rotation
            _delay_ms(200);  // Debounce delay
        } else if (button_30_pressed()) {
            f30 = 1;
            f180 = f360 = 0;  // Reset other modes
            steps = 17;       // 30° rotation
            _delay_ms(200);   // Debounce delay
        }

        // Rotate stepper motor based on the selected mode
        if (f180) {
            rotate_forward(steps, step_delay);  // Continuous rotation for 180°
        } else if (f360) {
            rotate_forward(steps, step_delay);  // Continuous rotation for 360°
        } else if (f30) {
            rotate_forward(steps, step_delay);  // Continuous rotation for 30°
        }
    }
}
```