DEPARTMENT OF PHYSICS

UNIVERSITY OF COLOMBO

PH3032-EMBEDDED SYSTEM LABORATORY

Development of a 2D Plotter using Embedded Systems

Name : K.D.P.R. Jayawardena

Index no: s16036

Date : 24/10/2024

Table of Contents

Table of Figures

1. Abstract

Modern automation world requires robust machines which are mostly developed using embedded systems. So this report contains the design and development of a 2D plotter controlled by an ATmega328P microcontroller, aimed at precision plotting using G-code instructions. The system employs three NEMA 17 stepper motors for X and Y-axis movement and a servo motor to control the Z-axis for lifting the pen during plotting. Custom firmware was developed to interpret G-code commands and manage motion control, utilizing A4988 motor drivers and serial communication via the CH340 module. The firmware ensures accurate navigation, pen control, and synchronization of movements. Key features such as external interrupts for limit switches provide enhanced precision and safety, preventing motor overextension. The plotter demonstrated high accuracy in drawing complex designs, showcasing its potential for applications in technical drawings, artistic sketches, and rapid prototyping. This project exemplifies the fusion of embedded systems and mechanical hardware to achieve automation and efficiency, emphasizing the importance of microcontroller-based systems in modern engineering solutions.

2. Introduction

The field of automated drawing systems has seen significant advancements in recent years, with applications ranging from industrial manufacturing to personal hobbyist projects. Among these innovations, 2D plotters have emerged as versatile tools capable of transforming digital designs into physical outputs with precision and efficiency. This project focuses on the design, development, and implementation of a 2D plotter that utilizes the ATmega328P microcontroller, offering a cost-effective, efficient, and user-friendly solution for drawing and plotting tasks.

The core objective of this project is to create a system that can interpret G-code instructions to control two Nema 17 stepper motors for X and Y-axis movements, and a SG90 servo motor for Z-axis control, ensuring accurate pen lifting and placement. The ATmega328P microcontroller was selected for its balance of performance, simplicity, and affordability, making it ideal for hobbyists and small-scale applications. The plotter converts digital commands into precise physical actions, serving various purposes such as technical drawing, design prototyping, and educational demonstrations.

To achieve seamless operation, the firmware running on the ATmega328P microcontroller processes G-code commands received via UART serial communication from a Python-based interface. This firmware manages all motor control and pen movements, ensuring accurate reproduction of digital designs. The Nema 17 stepper motors, driven by A4988 motor drivers, offer high torque and precision for smooth motion, while the SG90 servo motor enables reliable pen control for intricate details. The use of G-code, a standard language in CNC systems, ensures compatibility with existing design software, expanding the plotter's potential applications.

This report covers the key stages of the project, including hardware design, firmware development, and software integration. It also discusses the challenges encountered, such as ensuring precise motor coordination and optimizing communication protocols, as well as the solutions devised to overcome these obstacles. Through this project, we aim to demonstrate how a simple, yet powerful system can be created using accessible and affordable technology, making 2D plotting capabilities available to a wider audience.

3. Methodology

The ATmega328P microcontroller is utilized in the design and implementation of an advanced 2D plotter system. This project leverages the microcontroller's precision and functionality to develop a robust, accurate plotting machine, ensuring precise movement control and reliable performance through meticulous firmware development and rigorous testing of motor coordination and pen operations.

3.1 Mechanical Features

The ATmega328P microcontroller is utilized in the design and implementation of an advanced 2D plotter system. This project leverages the precision of the microcontroller and functionality to develop a robust, accurate plotting machine, ensuring precise movement control and reliable performance through meticulous firmware development and rigorous testing of motor coordination and pen operations. Mainly the 2D plotter consists of 3 Nema17 stepper motors, SG90 servo motor, thread bars, bearings, and a steady frame to hold all these mechanical items.

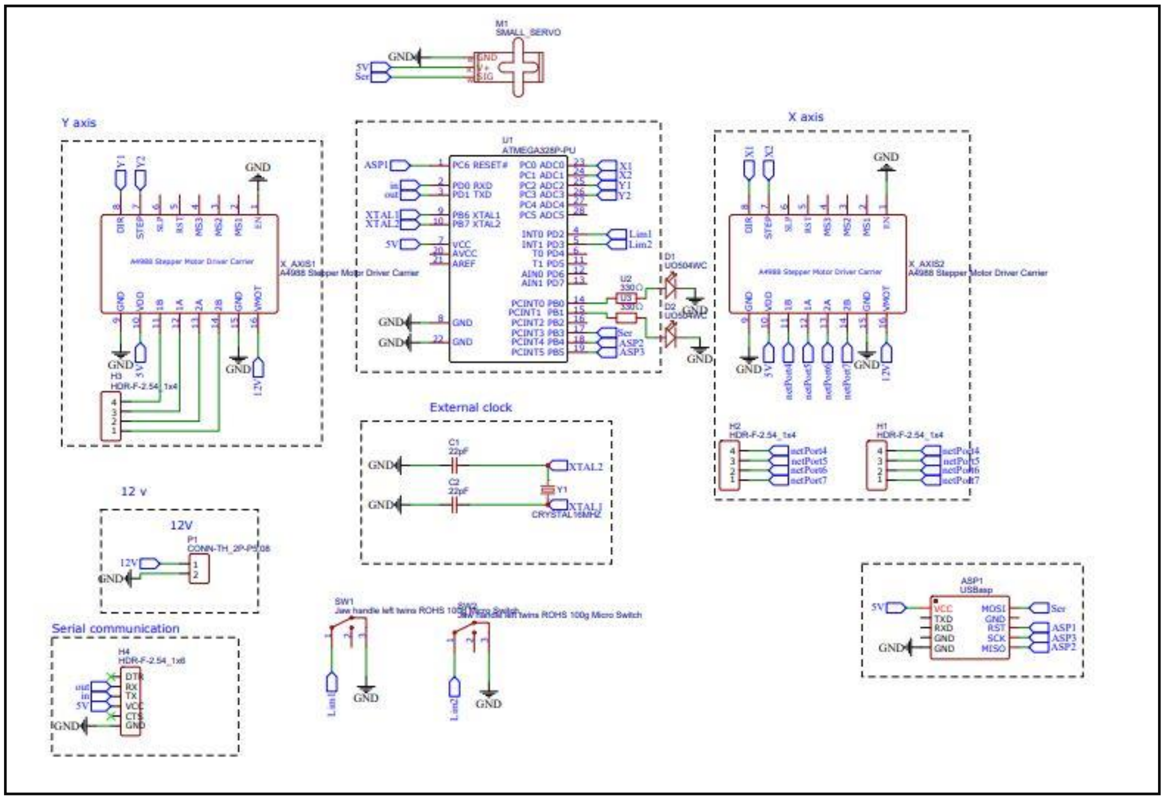

Figure 1: Schematic diagram of the circuit

3.2 UART Serial Communication

The core of the 2D plotter firmware lies in the efficient management of UART (Universal Asynchronous Receiver-Transmitter) serial communication. This component of the firmware is crucial as it establishes a seamless connection between the computer and the plotter, facilitating the accurate transmission of G-code instructions that control motor movements and pen operations. UART serial communication ensures that data is transmitted with minimal delay and error, providing the reliability required for precision plotting.

The firmware interprets these G-code commands and translates them into actionable signals for the stepper motors controlling the X and Y axes, as well as the servo motor managing the Z-axis (pen lift). The implementation of real-time feedback within the UART communication protocol also enables the system to correct any discrepancies, ensuring the plotter maintains high accuracy throughout the drawing process.

The following sections provide a comprehensive overview of the firmware development process, with a primary focus on UART serial communication, motor control logic, and the strategies employed to optimize communication efficiency and plotting precision. These sections also explore the challenges encountered in establishing stable serial communication and the solutions adopted to ensure smooth and reliable operation.

3.2.1 Initializing UART Serial Communication

This section sets up the hardware for serial communication between the 2D plotter and a computer, allowing the plotter to receive G-code commands.

a.) Setting baud rate

UART communication requires a defined speed or baud rate, the number of bits transmitted per second. This rate must be the same on both the transmitting (computer) and receiving (plotter) ends. The baud rate is configured using the UBRR (USART Baud Rate Register), which is split into UBRRL (low byte) and UBRRH (high byte).

```
UBRRL = BAUD_PRESCALE;
UBRRH = (BAUD_PRESCALE >> 8);
```

Figure 2:Setting baud rate

The BAUD_PRESCALLER is a calculated value based on the system clock frequency (F_CPU) and the desired baud rate. This value ensures that the UART module operates at the right communication speed.

b.) Enabling UART transmitter and receiver

To facilitate serial communication, both the transmitter (for sending data) and the receiver (for receiving data) must be enabled. The UCSRB (USART Control and Status Register B) is configured to enable the following functionalities:

- RXEN (Receiver Enable) turns on the receiver module, allowing the plotter to listen to incoming data.
- TXEN (Transmitter Enable) activates the transmitter module, enabling the plotter to send data back to the computer if necessary.

```
UCSRB = (1 << RXEN) | (1 << TXEN);
```

Figure 3: Enabling UART transmitter and receiver

c.) Setting frame format

The frame format defines how the bits are structured in each transmission. It includes settings for the number of data bits, stop bits, and parity checking. The UCSRC (USART Control and Status Register C) is used to set the frame format.

```
UCSRC = (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1);
```

Figure 4:Setting frame format

URSEL (Register Select) is set to 1 to indicate that we are modifying the UCSRC register. UCSZ1 and UCSZ0 are both set to 1, configuring the frame size to 8 data bits. The default frame format (as configured) also includes 1 stop bit and no parity bit.

The code is setting up UART in asynchronous mode, which means that no external clock signal is required for communication. Instead, both the transmitter and receiver operate independently based on their internal clocks, synchronized by the baud rate. Although not directly initialized in the init_uart() function, UART communication involves sending and receiving data through the UDR (USART Data Register). When receiving data, the hardware stores incoming characters in a buffer, which can later be read into your program by accessing the UDR register.

3.3 Parsing and processing of data

The code handles data parsing and command processing through key functions designed to read, interpret, and execute G-code commands. G-code is the language used to instruct the plotter on moving and controlling motors, including directions, distances, and the servo for the Z-axis.

3.3.1 Reading G-code input (read_line_from_serial function)

This function continuously reads characters from the serial input (UART) into a buffer. Once a newline (\n) or carriage return (\r) character is detected, it indicates the end of the command, and the buffer is marked as ready for processing (buffer_ready = 1).

```c
// Read a line of G-code from the serial input
void read_line_from_serial(char *buffer)
{
    int buffer_index = 0;  // Start at the beginning of the
buffer

    while (1)
    {
        // Wait for data to be received
        while (!(UCSRA & (1 << RXC)));

        // Get the received data and store it in the buffer
        char c = UDR;
        buffer[buffer_index++] = c;

        // Check for end of line (newline or carriage return)
        if (c == '\n' || c == '\r')
        {
            buffer[buffer_index] = '\0';  // Null-terminate the
string
            buffer_index = 0;
            buffer_ready = 1;  // Set the flag to indicate the
buffer is ready
            break;
        }
    }
}
```

Figure 5:read_line_from_serial function

3.3.2 Tokenizing G-code commands (split function)

The split function takes the buffer (containing the received G-code command) and breaks it into smaller parts, called tokens. Each token is typically a command like X, Y, Z, or F followed by values representing distances, feed rate, or angles. strtok is used to break the buffer into tokens separated by spaces, storing each token in the gcodeArr[] array for processing.

```c
// Split the received G-code line into individual tokens
void split(char *buffer)
{
    int i = 0;
    char *token = strtok(buffer, " "); // Tokenize the string by
spaces
    while (token != NULL && i < MAX_TOKENS)
    {
        gcodeArr[i++] = token;  // Store the token in the G-code
array
        token = strtok(NULL, " "); // Get the next token
    }
}
```

Figure 6:split function

3.3.3 Processing G-code Commands (process_command function)

This is where the core interpretation of the G-code happens. The function loops through the gcodeArr[] array, checking each token to determine the type of command (e.g. move along the X-axis, Y-axis, change feed rate, or control the servo motor).

The key commands handled are:

- X and Y: Move the stepper motors by calculating the distance from the current position (*x or *y) and updating the distances (x_dist, y_dist).
- F: Set the feed rate (speed of movement) by updating the feed variable.
- Z: Control the servo motor (pen up/down) by adjusting the servo_angle and calling set_servo_angle to physically move the servo to the specified angle.

For each movement command (X, Y), the new position is calculated, and the motors are instructed to move by the calculated distance.

```
// Process the G-code commands and set the distances for
movement or control actions
void process_command(float *x, float *y, char *gcodeArr[])
{
    x_dist = 0;  // Reset X distance
    y_dist = 0;  // Reset Y distance
    char *ptr;
    for (int i = 0; gcodeArr[i] != NULL; i++)
    {
        switch (gcodeArr[i][0])
        {
        default:
            break;
        case 'X':  // G-code command to move in the X direction
            x_dist = strtod(gcodeArr[i] + 1, &ptr) - *x;  //
Calculate the distance to move in X
            *x += x_dist;  // Update the current X position
            break;
        case 'Y':  // G-code command to move in the Y direction
            y_dist = strtod(gcodeArr[i] + 1, &ptr) - *y;  //
Calculate the distance to move in Y
            *y += y_dist;  // Update the current Y position
            break;
        case 'F':  // G-code feed rate command
            feed = strtol(gcodeArr[i] + 1, &ptr, 10);  // Update
feed rate (speed)
            break;
        case 'Z':  // G-code command to lift or lower the pen (Z
axis control)
            servo_angle = strtod(gcodeArr[i] + 1, &ptr);  //
Update the servo angle (pen up/down)
            set_servo_angle(servo_angle);  // Move the servo to
the specified angle
            break;
        }
    }
}
```

Figure 7:process_command function

3.3.4 Moving motors (move_x_motor and move_y_motor functions)

These functions physically control the stepper motors based on the distances calculated in process_command.

The functions convert the distance to steps, adjust the direction of the motor, and then perform the required number of steps, controlling the speed of movement using the delay_time variable (calculated from the feed rate).

```c
// Move the X-axis motor by the specified distance
void move_x_motor(float x_dist, int delay_time)
{
    int steps = fabs(x_dist) * stpmm;  // Calculate the number
of steps to move
    if (x_dist > 0)
        PORTA |= dir_x;  // Set direction for forward movement
    else if (x_dist < 0)
        PORTA &= ~dir_x; // Set direction for backward movement

    // Perform the movement by toggling the step pin for each
step
    for (int i = 0; i < steps; i++)
    {
        PORTA |= stp_x;
        _delay_us(100);
        PORTA &= ~stp_x;
        _delay_ms(delay_time);
    }
}
```

Figure 8:Moving motors

3.3.5 Servo Control (set_servo_angle function)

The following given function controls the up and down movement of the pen by setting the servo motor angle.

```c
// Set the servo motor to the specified angle
void set_servo_angle(int angle)
{
    OCR0 = ((angle * 0.0555) + 17);  // Set the Output Compare
Register to adjust the PWM duty cycle
}
```

Figure 9:set_servo_angle function

By breaking down the G-code into tokens, processing them into movement commands, and controlling the motors and servo, the machine can move to specified coordinates, adjust the feed rate, and perform pen up/down actions

3.4 Executing processed data

The execution of the processed G-code data in the provided code involves translating the parsed instructions into motor movements and servo control actions.

3.4.1 Processing the G-code instructions

The G-code data is parsed and stored in the array gcodeArr (via the split function). After splitting, each token (e.g., G1, X10, Y20, F600, Z90) is processed by the process_command function. This function extracts useful information like the movement distances (X and Y), feed rate (F), and pen control (Z).

For example, from the G-code G1 X10 Y20 F600 Z90:

- G1: Specifies a linear movement command
- X10: Indicates moving 10 mm along the X-axis
- Y20: Indicates moving 20 mm along the Y-axis
- F600: Sets the feed rate to 600 mm/min
- Z90: Sets the servo to 90 degrees (lift the pen)

3.4.2 Executing X and Y movements

Once the movement commands (e.g., X10, Y20) are processed, the corresponding move_x_motor() and move_y_motor() functions are called to physically move the stepper motors. These functions take the calculated movement distances and generate the required step pulses to drive the motors.

```
// Move the X-axis motor by the specified distance
void move_x_motor(float x_dist, int delay_time)
{
    int steps = fabs(x_dist) * stpmm;  // Calculate the number
of steps to move
    if (x_dist > 0)
        PORTA |= dir_x;  // Set direction for forward movement
    else if (x_dist < 0)
        PORTA &= ~dir_x; // Set direction for backward movement

    // Perform the movement by toggling the step pin for each
step
    for (int i = 0; i < steps; i++)
    {
        PORTA |= stp_x;
        _delay_us(100);
        PORTA &= ~stp_x;
        _delay_ms(delay_time);
    }
}
```

Figure 10:Executing X and Y movements

- Steps calculation: The number of steps required to move the X-axis by a certain distance is calculated using steps = fabs(x_dist) * stpmm, where stpmm is the number of steps required to move 1 mm (defined based on your motor's step size and driver configuration).
- Step generation: For each calculated step, the stp_x pin is toggled (set high and then low), which sends step pulses to the motor driver, causing the motor to rotate by one step. The delay_time controls the speed of these pulses, which is related to the feed rate.
- Y-Axis movement (move_y_motor) works similarly, but with its own step and direction pins.
- Direction setup: Depending on whether x_dist is positive or negative, the motor direction is set:
    - For positive movement, PORTA |= dir_x sets the motor direction to forward.
    - For negative movement, PORTA &= ~dir_x sets the motor direction to backward.

3.4.3 Controlling the feed rate

The feed rate (F600) controls the speed of the motor movements. In the code, the feed value is processed as part of the G-code and used to control the delay between each step pulse in move_x_motor() and move_y_motor(). A higher feed rate will reduce the delay between pulses, making the motor move faster. The delay is managed using _delay_ms(delay_time), and delay_time is adjusted based on the feed rate value extracted from the G-code.

3.4.4 Executing Z-axis (servo control)

The Z-axis is managed by the servo motor, which is controlled using PWM (Pulse Width Modulation). The G-code command Z90 means the servo should be set to 90 degrees, which translates to lifting the pen. This function adjusts the servo motor position by setting its angle, which determines whether the pen is up or down. The desired angle (typically 0 degrees for pen down and 90 degrees for pen up) is converted into a PWM signal by calculating the value for the Output Compare Register (OCR0). This value changes the duty cycle of the PWM signal, controlling the servo motor's position. Accurate pen control is essential for lifting the pen at the correct points during plotting

```
// Set the servo motor to the specified angle
void set_servo_angle(int angle)
{
    OCR0 = ((angle * 0.0555) + 17);  // Set the Output Compare
Register to adjust the PWM duty cycle
}
```

Figure 11:servo control

3.4.5 initTimer0PWM(void) function

The initTimer0PWM() function initializes Timer0 in PWM (Pulse Width Modulation) mode to control the servo motor on the Z-axis. By configuring the timer registers (TCCR0), it sets the timer in fast PWM mode, allowing for fine control over the duty cycle of the PWM signal. The PWM signal controls the angle of the servo motor, which lifts or lowers the pen as needed (depending on the Z command in G-code). This function is crucial for controlling the pen's position with precision.

```
// Initialize Timer0 for PWM control (for controlling the servo
motor)
void initTimer0PWM(void)
{
    DDRB |= servoPin;  // Set the servo pin as output
    TCCR0 |= (1 << WGM00) | (1 << WGM01) | (1 << COM01) | (1 <<
CS01);  // Set up Timer0 in fast PWM mode
}
```

Figure 12:initTimer0PWM(void) function

3.5 External Interrupts

External interrupts are triggered by external events, such as changes in the state of a pin. In microcontrollers like the ATmega328P, external interrupts allow the microcontroller to respond immediately to external stimuli, such as button presses, sensor outputs, or signal transitions. External interrupts are often used in applications requiring immediate attention to real-world events without constantly polling the inputs.

External interrupts for the X-axis limit switch (connected to INT0), the Y-axis limit switch (connected to INT1), and the stop button (connected to INT2) are initialized. The General Interrupt Control Register (GICR) is configured to enable interrupts on INT0, INT1, and INT2.

3.5.1 INT0 (Limit switch for X-axis)

- Pin: PD3
- Trigger: Falling edge (when the limit switch is pressed or activated, i.e., when the signal transitions from HIGH to LOW)
- Usage: This interrupt is used to detect when the X-axis of the plotter reaches its limit. A limit switch is placed at the extreme end of the X-axis to prevent the stepper motor from moving beyond the allowed boundary.

3.5.2 INT1 (Limit switch for Y-axis)

- Pin: PD3
- Trigger: Falling edge (similar to INT0, triggered when the Y-axis limit switch is pressed)
- Usage: This interrupt is used to detect when the Y-axis of the plotter reaches its limit. The limit switch for the Y-axis is placed at its extreme position to stop or adjust movement and prevent the motor from moving out of bounds.

3.5.3 INT2 (Stop button)

- Pin: PB2
- Trigger: Rising edge (when the stop button is pressed and the signal transitions from LOW to HIGH)
- Usage: This interrupt is used to handle an emergency stop button. When the stop button is pressed, the interrupt is triggered, and the system should stop all movements immediately. The stopLED is turned on to indicate that the stop button has been pressed.

```
// Initialize external interrupts for limit switches and stop
button
void externalInit(void)
{
    // Enable external interrupts INT0 (X-axis limit), INT1 (Y-
axis limit), and INT2 (stop button)
    GICR |= (1 << INT0) | (1 << INT1) | (1 << INT2);

    // Set interrupt triggers: INT0 and INT1 on falling edge,
INT2 on rising edge
    MCUCR |= (1 << ISC01) | (1 << ISC11) | (1 << ISC21);
}

// External interrupt handler for the stop button (INT2)
ISR(INT2_vect)
{
    PORTB |= stopLED;  // Turn on the stop LED to indicate a
stop condition
}
```

Figure 13:External interrupts

4. Results and Analysis

4.1 Performance of the 2D Plotter

The 2D plotter, driven by the ATmega328P microcontroller, successfully performs the task of drawing shapes and patterns based on G-code inputs. The following results were observed during testing:

1. Plotting Accuracy:
   - The plotter demonstrated high accuracy in following the G-code commands. It could draw precise lines, curves, and shapes with minimal deviation. The accuracy was dependent on the resolution of the stepper motors and the precision of the G-code input.
   - Observation: The X-axis and Y-axis movements were consistent with the expected positions, allowing the pen to draw without visible misalignments.
2. Speed of Operation:
   - The plotter operated at a reasonable speed, capable of completing standard drawings within a few minutes. Adjustments to motor speed were made through step delays, balancing speed and accuracy.
   - Analysis: Increasing the speed beyond a certain threshold resulted in loss of stepper motor accuracy, leading to errors in the plotted image. A moderate speed setting provided the best trade-off between speed and accuracy.
3. Pen Control (Z-axis):
   - The servo motor controlling the up-and-down movement of the pen performed well, raising and lowering the pen at the correct points in the drawing. This functionality was essential for creating images with gaps and non-continuous lines.
   - Observation: The pen consistently lifted when required and contacted the drawing surface when expected. Minor adjustments to the servo angle range ensured optimal pen positioning.
4. Limit Switch Functionality:
   - The limit switches on both X and Y axes successfully prevented the stepper motors from moving beyond their physical limits, ensuring no damage to the hardware.

    - Analysis: The use of interrupts for the limit switches provided immediate feedback, effectively stopping the motors when the boundary was reached. This safety feature worked as intended.

5. Stop Button Performance:
    - The emergency stop button worked effectively, instantly halting all operations when pressed. The stop LED indicator was triggered, confirming the activation of the stop function.
    - Observation: Upon pressing the stop button, the system halted all motor operations, allowing for quick intervention if necessary. This feature is crucial for safety during testing or in case of system malfunction.

4.2 Analysis of G-code Interpretation

The G-code interpreter, implemented in the microcontroller, correctly parsed and executed G-code instructions. The following analysis was made:

- Line and Arc Commands:
    - The plotter accurately followed G01 (linear motion) and G02/G03 (circular interpolation) commands, drawing straight lines and arcs as per the input G-code. These movements were smoothly translated into stepper motor control signals.
    - Analysis: The calculation of motor steps based on the G-code coordinates and geometry (for arcs) was verified to be accurate within a reasonable tolerance level. Minimal discrepancies occurred, primarily due to step resolution limits of the motors.
- Pen Lift Commands:
    - The pen lifting (M05) and lowering (M03) commands were executed with precise timing, ensuring that the pen lifted during non-drawing moves and contacted the surface during drawing moves.
    - Observation: The transition between pen up and down was smooth and corresponded perfectly to the G-code input, ensuring the correct placement of lines and gaps.

4.3 Limitations and Improvements

While the overall performance of the 2D plotter was satisfactory, some limitations were observed:

1. Stepper Motor Precision:
    - The stepper motors used in the plotter had limited resolution, which occasionally led to minor deviations in very fine or highly detailed drawings. Increasing the motor step resolution or using micro-stepping could improve accuracy.
2. Mechanical Stability:
    - Minor vibrations in the mechanical structure occasionally affected drawing precision. Reinforcing the plotter frame or using higher-quality materials would reduce these effects.
3. Speed-Accuracy Trade-off:
    - Operating the plotter at higher speeds led to reduced accuracy due to the inability of the stepper motors to keep up with fast step commands. Optimizing the speed settings further or using more powerful motors could mitigate this issue.
4. Pen Lifting Mechanism:
    - The servo motor controlling the pen lifting was precise but had a slight delay, which could be improved for faster transitions between pen-up and pen-down states.

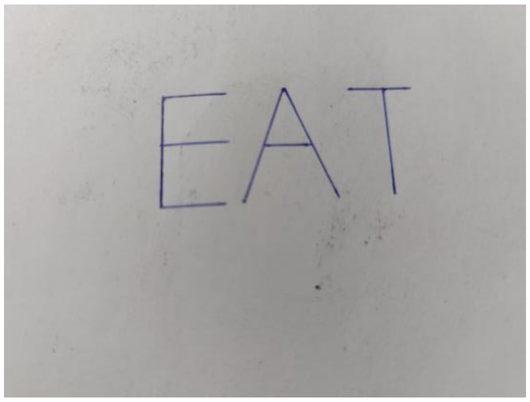

Figure 14: Image of a result from 2d plotter

5. Discussion

The design and development of the 2D plotter offers a range of insights into the challenges and considerations involved in creating a precise, automated drawing machine. The following points discuss the key aspects of the project, highlighting its strengths, challenges, and potential improvements.

5.1 Mechanical and Electrical Design

The mechanical design of the plotter, involving a pen mounted on a movable platform controlled by stepper motors, proved to be effective for producing 2D drawings. The choice of using stepper motors for both the X and Y axes allowed for precise positioning of the pen, which is essential for accurate drawing.

- Strengths:
    - The stepper motors provided accurate and repeatable movement due to their precise control of steps, allowing the plotter to draw detailed shapes.
    - The use of a servo motor for the Z-axis (pen lift) ensured that the pen could be moved up and down reliably, which was crucial for drawing discontinuous lines and shapes.
- Challenges:
    - Mechanical vibrations from the motors sometimes affected the precision of the plotter, particularly when moving at higher speeds. This led to slight inaccuracies in the final drawings.
    - The overall stability of the machine is important for maintaining consistent accuracy, and improvements in the rigidity of the plotter structure could enhance performance.

5.2 Stepper Motor Control and G-code Execution

One of the core elements of the operation of the plotter was the control of the stepper motors based on G-code commands. The G-code interpreter was responsible for converting drawing instructions into motor movements.

- Strengths:
    - The plotter accurately interpreted and executed basic G-code commands such as G01 (linear movement) and G02/G03 (arc movement). This allowed for the creation of both straight lines and curved shapes.

    - The integration of limit switches ensured that the motors did not exceed their range of motion, preventing mechanical damage and improving operational safety.
- Challenges:
    - At higher speeds, stepper motor performance diminished slightly, leading to missed steps and reduced drawing accuracy. This is a common trade-off in stepper motor systems, where higher speed can compromise precision.
    - The use of G-code limited the complexity of the drawings. While the plotter could execute simple shapes, it lacked advanced features like dynamic speed adjustment, acceleration control, and toolpath optimization, which could improve both speed and precision.

5.3 Software and Interrupts

The control system of the plotter relied heavily on software interrupts, particularly for handling limit switches and the emergency stop button. The use of interrupts enabled the system to respond quickly to critical events, improving both safety and control.

- Strengths:
    - The interrupt-based system allowed for immediate motor shutdown when the emergency stop button was pressed or when a limit switch was triggered. This quick response is essential for protecting both the machine and the workpiece.
    - The software effectively managed the plotting process, from reading G-code to executing motor movements, showing the flexibility and power of the ATmega328P microcontroller for real-time control tasks.
- Challenges:
    - The reliance on interrupts for critical events is effective but can introduce complexity into the software, especially if multiple interrupts occur simultaneously. Proper management and prioritization of interrupts are crucial for ensuring smooth operation.
    - Future versions of the system could benefit from more advanced software features like G-code parsing for complex commands and multi-axis coordination for smoother motion control.

5.4 Limitations and Potential Improvements

Despite the success of the 2D plotter, several limitations were identified during the testing phase. Addressing these limitations could significantly improve the performance and usability of the machine.

- Mechanical Stability: The current design could benefit from a more rigid frame to reduce vibrations and improve drawing precision. Reinforcing the structure or using higher-quality materials would enhance stability, particularly during high-speed operation.
- Motor Control: The stepper motors used in the plotter provided sufficient precision for basic drawings, but their resolution could be improved. Implementing micro-stepping would allow for finer control, enabling more detailed and accurate drawings.
- Pen Lift Speed: The pen lifting mechanism, controlled by a servo motor, was functional but could be optimized for faster transitions between up and down positions. This would reduce downtime between drawing strokes and improve overall speed.
- Advanced Features: The plotter could be enhanced by adding more advanced software features. For example, implementing acceleration and deceleration control for the motors would result in smoother motion, particularly at the beginning and end of each stroke. Additionally, support for more complex G-code commands could expand the plotter's functionality, allowing it to handle more intricate designs.

5.5 Comparison to Commercial Plotters

When compared to commercial 2D plotters, the DIY version built for this project offered many similar basic functions but at a significantly lower cost. However, commercial plotters often feature more advanced motor control systems, higher precision, and faster operation due to their use of advanced components and software.

- Cost vs. Functionality: The low cost of the DIY plotter makes it an attractive option for simple tasks such as drawing basic shapes and learning about CNC systems. However, for high-precision work or industrial applications, a commercial plotter with enhanced capabilities would be more suitable.

- User Interface: Commercial plotters typically offer more user-friendly interfaces, often including graphical displays or PC software for designing plots. In contrast, the DIY plotter relies on manually prepared G-code files, which may be less convenient for users without technical knowledge.

5.6 Practical Applications and Future Scope

The 2D plotter project has demonstrated its potential for various practical applications. For example, it could be used for prototyping, educational purposes, or even artistic drawing. There is significant room for improvement and expansion in the following areas:

- Multi-color Plotting: Future iterations could include multiple pens or tools, allowing the plotter to switch between different colors or drawing instruments.
- Larger Workspace: Expanding the physical dimensions of the plotter would allow for larger drawings, opening new possibilities for more extensive projects.
- 3D Plotter Evolution: With modifications, the 2D plotter could be expanded into a 3D plotting or even a basic 3D printing system. This would involve adding a third stepper motor for controlling vertical movement and adjusting the G-code interpreter to handle three-dimensional paths.

6. Conclusion

The 2D plotter project, centered around the ATmega328P microcontroller, has proven to be an effective and versatile tool for precision plotting tasks. By employing NEMA 17 stepper motors and A4988 motor drivers, the system ensures precise motion control along both the X and Y axes, while a servo motor modulates the Z-axis to lift the pen at appropriate times. The system effectively reads and interprets G-code commands sent through serial communication, enabling the machine to plot intricate designs with high accuracy.

Through custom firmware, the 2D plotter can seamlessly manage motor control, coordinate system navigation, and the timing required for pen-lifting and plotting movements. The accuracy achieved by this system makes it suitable for a wide variety of applications, from artistic sketches to technical design plotting.

The integration of external interrupts, such as limit switches, further enhances the precision and safety of the device. These interrupts ensure that the motors operate within the desired boundaries, avoiding overextension and protecting the device from potential mechanical stress.

In conclusion, the 2D plotter represents a successful demonstration of combining hardware and embedded systems to achieve automation and precision. The ATmega328P microcontroller has played a pivotal role in this project, serving as the core that unifies various components for smooth operation. This project not only highlights the capabilities of embedded systems in real-world applications but also exemplifies how such systems can be tailored to innovate and drive efficiency in engineering tasks. The success of the 2D plotter underlines its potential for further development and adaptation to various plotting and prototyping needs.

7. References

1. (No date a) *(PDF) G-code controlled 2D robotic plotter - researchgate*. Available at: https://www.researchgate.net/publication/322244352_G-Code_Controlled_2D_Robotic_Plotter (Accessed: 24 October 2024).

2. (No date b) *What is the best way to generate GCode for 2D plotter? | Researchgate*. Available at: https://www.researchgate.net/post/What_is_the_best_way_to_generate_GCode_for_2D_plotter (Accessed: 24 October 2024).

3. *Arduino based mini CNC 2d Plotter* (no date) *Arduino Project Hub*. Available at: https://projecthub.arduino.cc/Mrinnovative/arduino-based-mini-cnc-2d-plotter-796c2f (Accessed: 24 October 2024).

4. *EasyDraw v2 writing and Drawing Machine (fully assembled)* (no date) *shopmakerq.com - DIY Robotics store*. Available at: https://shopmakerq.com/product/2d-cnc-plotter-a-machine-for-writing-and-drawing-fully-assembled/ (Accessed: 24 October 2024).

5. IJRITCC, I.J. (2018) *G-code controlled 2D robotic plotter*, *Academia.edu*. Available at: https://www.academia.edu/36847627/G-Code_Controlled_2D_Robotic_Plotter (Accessed: 24 October 2024).

6. Indira (no date) *Radware bot manager Captcha*. Available at: https://iopscience.iop.org/article/10.1088/1742-6596/1950/1/012012/pdf (Accessed: 24 October 2024).

7. innovative, M. and Instructables (2017) *How to make mini CNC 2D plotter using scrap DVD drive, L293D Motor Shield & Arduino*, *Instructables*. Available at: https://www.instructables.com/How-to-Make-Mini-CNC-2D-Plotter-Using-Scrap-DVD-Dr/ (Accessed: 24 October 2024).

8. Pilhuhn (no date) *Pilhuhn/XY-plotter: Arduino-code for my xy-plotter*, *GitHub*. Available at: https://github.com/pilhuhn/xy-plotter (Accessed: 24 October 2024).

9. Technologies, H. (2022) *Design and implementation of low-cost 2D plotter computer numeric control (CNC) machine*, *Academia.edu*. Available at: https://www.academia.edu/80500237/Design_and_Implementation_of_low_cost_2D_plotter_Computer_Numeric_Control_CNC_Machine (Accessed: 24 October 2024).

10. vishnu_, R. and Instructables (2020) *The Arduino CNC drawing machine: 2d plotter: How to make Arduino 2D plotter CNC Machine*, *Instructables*. Available at: https://www.instructables.com/The-Arduino-CNC-Drawing-Machine/ (Accessed: 24 October 2024).

8. Appendix

C code

```c
#define F_CPU 16000000UL  // CPU frequency is set to 16 MHz

#include <avr/io.h>       // Standard AVR I/O library
#include <util/delay.h>    // Delay functions
#include <avr/interrupt.h> // Interrupt handling
#include <stdio.h>         // Standard input/output
#include <string.h>        // String handling
#include <stdlib.h>        // Standard library functions (e.g., memory allocation)
#include <math.h>          // Math functions, used for floating-point operations

// Pin definitions for controlling motors and other components
#define servoPin (1 << PB3)  // Servo motor pin for Z-axis control
#define dir_x (1 << PA4)     // Direction pin for X-axis stepper motor
#define stp_x (1 << PA5)     // Step pin for X-axis stepper motor
#define dir_y (1 << PA6)     // Direction pin for Y-axis stepper motor
#define stp_y (1 << PA7)     // Step pin for Y-axis stepper motor
#define stopLED (1 << PB1)   // LED for indicating stop condition
#define onLED (1 << PB0)     // LED for indicating system on state
#define limit_x (1 << PD2)   // Limit switch for X-axis (interrupt pin INT0)
#define limit_y (1 << PD3)   // Limit switch for Y-axis (interrupt pin INT1)
#define stop (1 << PB2)      // Stop button (interrupt pin INT2)

#define MAX_TOKENS 30        // Maximum number of tokens for G-code parsing
#define stpmm 160            // Steps per millimeter for motors (with 1/16th step resolution)

// Global variables
char *gcodeArr[MAX_TOKENS];  // Array for storing G-code commands
float x;                     // Current X position
float y;                     // Current Y position
float x_dist;                // Distance to move on X-axis
float y_dist;                // Distance to move on Y-axis
int feed;                    // Feed rate (speed of movement)
int delay_time;              // Delay between steps based on feed rate
int servo_angle = 0;         // Servo angle (0 for pen down, 90 for pen up)
int BAUD_PRESCALE = 8;       // Baud rate prescaler for UART communication

// Buffer for storing received commands from serial
char buffer[64];             // Serial input buffer
int buffer_index = 0;        // Buffer index for serial input

// Flag to indicate when the buffer is ready for processing
volatile uint8_t buffer_ready = 0;  // Volatile flag for synchronization

// Function prototypes (declarations of functions used in the program)
```

```c
void init(void);                                // Initialize general settings
void init_motors(void);                         // Initialize stepper motors
void init_uart(void);                           // Initialize UART for serial communication
void read_line_from_serial(char *buffer);    // Read a line of G-code from serial input
void split(char *buffer);                       // Split G-code commands into tokens
void process_command(float *x, float *y, char *gcodeArr[]); // Process G-code commands
void move_x_motor(float x_dist, int delay_time); // Move the X-axis motor by the specified distance
void move_y_motor(float y_dist, int delay_time); // Move the Y-axis motor by the specified distance
void initTimer0PWM(void);                       // Initialize Timer0 for PWM control (for the servo motor)
void set_servo_angle(int angle);                // Set the servo motor to a specific angle
void externalInit(void);                        // Initialize external interrupts

// Main function
int main(void)
{
    init();                      // Initialize general settings
    sei();                       // Enable global interrupts
    externalInit();               // Initialize external interrupts for limit switches
    init_uart();                  // Initialize UART for receiving G-code commands
    initTimer0PWM();              // Initialize Timer0 for PWM to control the servo motor
    PORTB |= onLED;               // Turn on the system LED to indicate the system is ready

    // Main loop
    while (1)
    {
        // Set stepper motor pins to high impedance mode (input mode)
        DDRA &= (~(dir_x|stp_x|dir_y|stp_y));
        PORTA &= (~(dir_x|stp_x|dir_y|stp_y));

        // Read G-code line from serial input
        read_line_from_serial(buffer);

        // Process the buffer if it is ready (i.e., full G-code command received)
        if (buffer_ready)
        {
            split(buffer);                          // Split the received G-code line into tokens
            process_command(&x, &y, gcodeArr);      // Process the G-code to determine motor movements and actions

            // Set delay time based on the feed rate (speed of movement)
            switch (feed)
            {
                default:
                    break;
                case 3000:  // Fast movement
                    delay_time = 0;
                    break;
                case 1000:  // Medium speed
                    delay_time = 2;
                    break;
                case 600:   // Slow movement
                    delay_time = 4;
                    break;
            }

            // Move the motors based on the parsed G-code
            move_x_motor(x_dist, delay_time);   // Move X-axis by the calculated distance
            move_y_motor(y_dist, delay_time);   // Move Y-axis by the calculated distance

            // Clear the buffer and reset the buffer index and flag for the next command
            memset(buffer, 0, sizeof(buffer));
            buffer_index = 0;
            buffer_ready = 0;
        }
    }

    return 0;
}

// Initialize general settings
void init(void)
{
    DDRB &= ~stop;  // Set stop button pin as input
    DDRB |= servoPin | onLED | stopLED;  // Set servo, onLED, and stopLED pins as output
    DDRD &= ~(limit_x | limit_y);  // Set limit switch pins as input
}

// Initialize stepper motors
void init_motors(void)
{
    DDRA |= dir_x | stp_x | dir_y | stp_y;  // Set direction and step pins for both X and Y motors as output
}

// Initialize UART for serial communication
void init_uart(void)
{
    UBRRL = BAUD_PRESCALE;       // Set the lower 8 bits of the baud rate value
    UBRRH = (BAUD_PRESCALE >> 8);// Set the upper 8 bits of the baud rate value

    UCSRB = (1 << RXEN) | (1 << TXEN); // Enable receiver and transmitter

    // Set frame format: 8 data bits, 1 stop bit, no parity
    UCSRC = (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1);
}

// Read a line of G-code from the serial input
void read_line_from_serial(char *buffer)
{
    int buffer_index = 0;  // Start at the beginning of the buffer

    while (1)
    {
        // Wait for data to be received
        while (!(UCSRA & (1 << RXC)));

        // Get the received data and store it in the buffer
        char c = UDR;
        buffer[buffer_index++] = c;

        // Check for end of line (newline or carriage return)
        if (c == '\n' || c == '\r')
        {
            buffer[buffer_index] = '\0';  // Null-terminate the string
            buffer_index = 0;
            buffer_ready = 1;  // Set the flag to indicate the buffer is ready
            break;
        }
    }
}

// Split the received G-code line into individual tokens
void split(char *buffer)
{
    int i = 0;
    char *token = strtok(buffer, " "); // Tokenize the string by spaces
    while (token != NULL && i < MAX_TOKENS)
    {
        gcodeArr[i++] = token;  // Store the token in the G-code array
        token = strtok(NULL, " "); // Get the next token
    }
}

// Process the G-code commands and set the distances for movement or control actions
void process_command(float *x, float *y, char *gcodeArr[])
{
    x_dist = 0;  // Reset X distance
    y_dist = 0;  // Reset Y distance
    char *ptr;
    for (int i = 0; gcodeArr[i] != NULL; i++)
    {
        switch (gcodeArr[i][0])
        {
        default:
            break;
        case 'X':  // G-code command to move in the X direction
```

```
            x_dist = strtod(gcodeArr[i] + 1, &ptr) - *x;  // Calculate the distance to move in X
            *x += x_dist;  // Update the current X position
            break;
        case 'Y':  // G-code command to move in the Y direction
            y_dist = strtod(gcodeArr[i] + 1, &ptr) - *y;  // Calculate the distance to move in Y
            *y += y_dist;  // Update the current Y position
            break;
        case 'F':  // G-code feed rate command
            feed = strtol(gcodeArr[i] + 1, &ptr, 10);  // Update feed rate (speed)
            break;
        case 'Z':  // G-code command to lift or lower the pen (Z axis control)
            servo_angle = strtod(gcodeArr[i] + 1, &ptr);  // Update the servo angle (pen up/down)
            set_servo_angle(servo_angle);  // Move the servo to the specified angle
            break;
        }
    }
}

// Move the X-axis motor by the specified distance
void move_x_motor(float x_dist, int delay_time)
{
    int steps = fabs(x_dist) * stpmm;  // Calculate the number of steps to move
    if (x_dist > 0)
        PORTA |= dir_x;  // Set direction for forward movement
    else if (x_dist < 0)
        PORTA &= ~dir_x; // Set direction for backward movement

    // Perform the movement by toggling the step pin for each step
    for (int i = 0; i < steps; i++)
    {
        PORTA |= stp_x;
        _delay_us(100);
        PORTA &= ~stp_x;
        _delay_ms(delay_time);
    }
}

// Move the Y-axis motor by the specified distance
void move_y_motor(float y_dist, int delay_time)
{
    int steps = fabs(y_dist) * stpmm;  // Calculate the number of steps to move
    if (y_dist > 0)
        PORTA |= dir_y;  // Set direction for forward movement
    else if (y_dist < 0)
        PORTA &= ~dir_y; // Set direction for backward movement

    // Perform the movement by toggling the step pin for each step
    for (int i = 0; i < steps; i++)
    {
        PORTA |= stp_y;
        _delay_us(100);
        PORTA &= ~stp_y;
        _delay_ms(delay_time);
    }
}

// Initialize Timer0 for PWM control (for controlling the servo motor)
void initTimer0PWM(void)
{
    DDRB |= servoPin;  // Set the servo pin as output
    TCCR0 |= (1 << WGM00) | (1 << WGM01) | (1 << COM01) | (1 << CS01);  // Set up Timer0 in fast PWM mode
}

// Set the servo motor to the specified angle
void set_servo_angle(int angle)
{
    OCR0 = ((angle * 0.0555) + 17);  // Set the Output Compare Register to adjust the PWM duty cycle
}

// Initialize external interrupts for limit switches and stop button
void externalInit(void)
{
    // Enable external interrupts INT0 (X-axis limit), INT1 (Y-axis limit), and INT2 (stop button)
    GICR |= (1 << INT0) | (1 << INT1) | (1 << INT2);

    // Set interrupt triggers: INT0 and INT1 on falling edge, INT2 on rising edge
    MCUCR |= (1 << ISC01) | (1 << ISC11) | (1 << ISC21);
}

// External interrupt handler for the stop button (INT2)
ISR(INT2_vect)
{
    PORTB |= stopLED;  // Turn on the stop LED to indicate a stop condition
}

```

Python code

```
import serial
import time

# Initialize global variables for x and y positions, feed rate, and pen state
x = 0
y = 0
feed = 1
pen_state = 0  # 0 = pen up, 1 = pen down
pen_delay = 0.2  # Delay when lowering or lifting pen

# Function to send G-code to the plotter
def send_gcode_file(filename, port, baudrate):
    global x, y, feed, pen_state, pen_delay

    # Open the serial port
    ser = serial.Serial(port, baudrate)
    if not ser.isOpen():
        ser.open()

    # Open the G-code file
    with open(filename, 'r') as f:
        # Read and process each line of G-code
        for line in f:
            ser.write(line.encode())  # Send G-code to plotter
            ser.write("\n".encode())  # Send newline
            print(line)  # Print the G-code line for debugging

            gcodeArr = line.split()  # Split G-code line into commands
            x_dist = 0
            y_dist = 0

            # Parse each command in the line
            for code in gcodeArr:
                if code[0] == 'X':  # Move to new X position
                    x_dist = float(code[1:]) - x
                    x += x_dist
                elif code[0] == 'Y':  # Move to new Y position
                    y_dist = float(code[1:]) - y
                    y += y_dist
                elif code[0] == 'F':  # Set feed rate (movement speed)
                    feed = float(code[1:])
                elif code[0] == 'S':  # Set pen state (S1 = pen down, S0 =
pen up)
                    pen_state = int(code[1:])
                    if pen_state == 1:
                        print("Pen down")  # Pen down
                        time.sleep(pen_delay)  # Wait for the pen to lower
                    else:
                        print("Pen up")  # Pen up
                        time.sleep(pen_delay)  # Wait for the pen to raise

            # Calculate movement delay based on distance and feed rate
            movement_delay = (abs(x_dist + y_dist) / (feed / 60))
            time.sleep(movement_delay * 1.05)  # Add small buffer to delay
    # Close the serial port when done
    ser.close()

# Example G-code file path
filename = 'C:\Users\Dell\Desktop\2d plotter.gcode'
send_gcode_file(filename, 'COM5', 57600)
```