```python
import tkinter as tk
from tkinter import ttk, messagebox, scrolledtext
import serial
import threading
import time
from datetime import datetime
import math

class PelletDispenserGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Pellet Dispenser Control")
        self.root.geometry("900x700")

        # Serial connection
        self.serial_port = None
        self.connected = False
        self.reading_thread = None
        self.running = True

        # Status variables
        self.system_running = tk.BooleanVar()
        self.refill_needed = tk.BooleanVar()
        self.cal_weight = tk.StringVar(value="0.0")
        self.pour_weight = tk.StringVar(value="0.0")
        self.time_between = tk.StringVar(value="0")
        self.hopper_weight = tk.StringVar(value="100.0")  # Default 100g capacity

        # New variables for hopper visualization
        self.current_hopper_weight = tk.DoubleVar(value=100.0)  # Current weight in hopper
        self.max_hopper_weight = tk.DoubleVar(value=100.0)  # Max capacity
        self.time_to_next_pour = tk.IntVar(value=0)  # Time remaining until next pour
        self.last_pour_time = None

        self.setup_gui()

    def setup_gui(self):
        # Main container
        main_frame = ttk.Frame(self.root, padding="10")
        main_frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

        # Connection frame
        conn_frame = ttk.LabelFrame(main_frame, text="Connection", padding="5")
        conn_frame.grid(row=0, column=0, columnspan=3, sticky=(tk.W, tk.E), pady=(0, 10))

        ttk.Label(conn_frame, text="COM Port:").grid(row=0, column=0, padx=(0, 5))
        self.port_var = tk.StringVar(value="COM3")
        port_combo = ttk.Combobox(conn_frame, textvariable=self.port_var, width=10)
        port_combo['values'] = self.get_serial_ports()
        port_combo.grid(row=0, column=1, padx=(0, 10))

        self.connect_btn = ttk.Button(conn_frame, text="Connect", command=self.toggle_connection)
        self.connect_btn.grid(row=0, column=2)

        self.status_label = ttk.Label(conn_frame, text="Disconnected", foreground="red")
        self.status_label.grid(row=0, column=3, padx=(10, 0))

        # Parameters frame
        params_frame = ttk.LabelFrame(main_frame, text="Parameters", padding="10")
        params_frame.grid(row=1, column=0, sticky=(tk.W, tk.E, tk.N), padx=(0, 10))

        # Calibration
        ttk.Label(params_frame, text="Calibration Weight (g):").grid(row=0, column=0, sticky=tk.W, pady=2)
        cal_entry = ttk.Entry(params_frame, textvariable=self.cal_weight, width=10)
        cal_entry.grid(row=0, column=1, sticky=(tk.W, tk.E), padx=(5, 0), pady=2)
        cal_btn = ttk.Button(params_frame, text="Set", command=lambda: self.send_command(f"cal {self.cal_weight.get()}"))
        cal_btn.grid(row=0, column=2, padx=(5, 0), pady=2)

        # Pour weight
        ttk.Label(params_frame, text="Pour Weight (g):").grid(row=1, column=0, sticky=tk.W, pady=2)
        pour_entry = ttk.Entry(params_frame, textvariable=self.pour_weight, width=10)
        pour_entry.grid(row=1, column=1, sticky=(tk.W, tk.E), padx=(5, 0), pady=2)
        pour_btn = ttk.Button(params_frame, text="Set", command=lambda: self.send_command(f"pour {self.pour_weight.get()}"))
        pour_btn.grid(row=1, column=2, padx=(5, 0), pady=2)

        # Time between pours
        ttk.Label(params_frame, text="Time Between (s):").grid(row=2, column=0, sticky=tk.W, pady=2)
        time_entry = ttk.Entry(params_frame, textvariable=self.time_between, width=10)
        time_entry.grid(row=2, column=1, sticky=(tk.W, tk.E), padx=(5, 0), pady=2)
        time_btn = ttk.Button(params_frame, text="Set", command=lambda: self.send_command(f"time {self.time_between.get()}"))
        time_btn.grid(row=2, column=2, padx=(5, 0), pady=2)

        # Hopper weight (max capacity)
        ttk.Label(params_frame, text="Hopper Capacity (g):").grid(row=3, column=0, sticky=tk.W, pady=2)
        hopper_entry = ttk.Entry(params_frame, textvariable=self.hopper_weight, width=10)
        hopper_entry.grid(row=3, column=1, sticky=(tk.W, tk.E), padx=(5, 0), pady=2)
        hopper_btn = ttk.Button(params_frame, text="Set", command=self.set_hopper_capacity)
        hopper_btn.grid(row=3, column=2, padx=(5, 0), pady=2)

        # Control frame
        control_frame = ttk.LabelFrame(main_frame, text="Control", padding="10")
        control_frame.grid(row=1, column=1, sticky=(tk.W, tk.E, tk.N), padx=(0, 10))

        self.start_btn = ttk.Button(control_frame, text="Start System", command=self.start_system)
        self.start_btn.grid(row=0, column=0, pady=2, sticky=(tk.W, tk.E))

        self.stop_btn = ttk.Button(control_frame, text="Stop System", command=self.stop_system)
        self.stop_btn.grid(row=1, column=0, pady=2, sticky=(tk.W, tk.E))

        ttk.Separator(control_frame, orient=tk.HORIZONTAL).grid(row=2, column=0, sticky=(tk.W, tk.E), pady=10)

        self.manual_btn = ttk.Button(control_frame, text="Manual Pour", command=self.manual_pour)
        self.manual_btn.grid(row=3, column=0, pady=2, sticky=(tk.W, tk.E))
```

```
        self.calibrate_btn = ttk.Button(control_frame, text="Calibrate", command=self.calibrate)
        self.calibrate_btn.grid(row=4, column=0, pady=2, sticky=(tk.W, tk.E))

        self.status_btn = ttk.Button(control_frame, text="Get Status", command=self.get_status)
        self.status_btn.grid(row=5, column=0, pady=2, sticky=(tk.W, tk.E))

        ttk.Separator(control_frame, orient=tk.HORIZONTAL).grid(row=6, column=0, sticky=(tk.W, tk.E), pady=10)

        self.alarm_on_btn = ttk.Button(control_frame, text="Alarm On", command=self.alarm_on)
        self.alarm_on_btn.grid(row=7, column=0, pady=2, sticky=(tk.W, tk.E))

        self.alarm_off_btn = ttk.Button(control_frame, text="Alarm Off", command=self.alarm_off)
        self.alarm_off_btn.grid(row=8, column=0, pady=2, sticky=(tk.W, tk.E))

        # Status frame
        status_frame = ttk.LabelFrame(main_frame, text="Status", padding="10")
        status_frame.grid(row=1, column=2, sticky=(tk.W, tk.E, tk.N))

        # Status indicators
        ttk.Label(status_frame, text="System Running:").grid(row=0, column=0, sticky=tk.W, pady=2)
        self.running_indicator = ttk.Label(status_frame, text="NO", foreground="red")
        self.running_indicator.grid(row=0, column=1, sticky=tk.W, padx=(10, 0), pady=2)

        ttk.Label(status_frame, text="Refill Needed:").grid(row=1, column=0, sticky=tk.W, pady=2)
        self.refill_indicator = ttk.Label(status_frame, text="NO", foreground="green")
        self.refill_indicator.grid(row=1, column=1, sticky=tk.W, padx=(10, 0), pady=2)

        # Current settings display
        ttk.Separator(status_frame, orient=tk.HORIZONTAL).grid(row=2, column=0, columnspan=2, sticky=(tk.W, tk.E), pady=10)

        ttk.Label(status_frame, text="Current Settings:", font=('TkDefaultFont', 9, 'bold')).grid(row=3, column=0, columnspan=2, sticky=tk.W, pady=(0, 5))

        self.settings_text = tk.Text(status_frame, height=8, width=25, font=('Courier', 9))
        self.settings_text.grid(row=4, column=0, columnspan=2, sticky=(tk.W, tk.E))

        # NEW: Hopper Visualization Frame
        self.create_hopper_display(main_frame)

        # Serial output frame
        output_frame = ttk.LabelFrame(main_frame, text="Serial Output", padding="5")
        output_frame.grid(row=3, column=0, columnspan=3, sticky=(tk.W, tk.E, tk.N, tk.S), pady=(10, 0))

        self.output_text = scrolledtext.ScrolledText(output_frame, height=12, width=80)
        self.output_text.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

        # Clear output button
        clear_btn = ttk.Button(output_frame, text="Clear Output", command=self.clear_output)
        clear_btn.grid(row=1, column=0, pady=(5, 0))

        # Configure grid weights
        self.root.columnconfigure(0, weight=1)
        self.root.rowconfigure(0, weight=1)
        main_frame.columnconfigure(0, weight=1)
        main_frame.columnconfigure(1, weight=1)
        main_frame.columnconfigure(2, weight=1)
        main_frame.rowconfigure(3, weight=1)  # Changed from row 2 to 3
        output_frame.columnconfigure(0, weight=1)
        output_frame.rowconfigure(0, weight=1)

        # Update settings display and hopper visualization
        self.update_settings_display()
        self.update_countdown_timer()

    def create_hopper_display(self, parent):
        """Create the visual hopper display"""
        hopper_frame = ttk.LabelFrame(parent, text="Hopper Status", padding="10")
        hopper_frame.grid(row=2, column=0, columnspan=3, sticky=(tk.W, tk.E), pady=(10, 10))

        # Create canvas for hopper visualization
        canvas_frame = ttk.Frame(hopper_frame)
        canvas_frame.grid(row=0, column=0, padx=(0, 20))

        self.hopper_canvas = tk.Canvas(canvas_frame, width=200, height=150, bg='white', relief='sunken', bd=2)
        self.hopper_canvas.pack()

        # Info frame next to canvas
        info_frame = ttk.Frame(hopper_frame)
        info_frame.grid(row=0, column=1, sticky=(tk.W, tk.N))

        # Weight remaining display
        ttk.Label(info_frame, text="Weight Remaining:", font=('TkDefaultFont', 10, 'bold')).grid(row=0, column=0, sticky=tk.W, pady=(0, 5))
        self.weight_remaining_label = ttk.Label(info_frame, text="100.0 g", font=('TkDefaultFont', 12), foreground="blue")
        self.weight_remaining_label.grid(row=1, column=0, sticky=tk.W, pady=(0, 10))

        # Fill percentage
        ttk.Label(info_frame, text="Fill Level:", font=('TkDefaultFont', 10, 'bold')).grid(row=2, column=0, sticky=tk.W, pady=(0, 5))
        self.fill_percentage_label = ttk.Label(info_frame, text="100%", font=('TkDefaultFont', 12), foreground="green")
        self.fill_percentage_label.grid(row=3, column=0, sticky=tk.W, pady=(0, 10))

        # Time until next pour
        ttk.Label(info_frame, text="Next Pour In:", font=('TkDefaultFont', 10, 'bold')).grid(row=4, column=0, sticky=tk.W, pady=(0, 5))
        self.next_pour_label = ttk.Label(info_frame, text="--", font=('TkDefaultFont', 12), foreground="orange")
        self.next_pour_label.grid(row=5, column=0, sticky=tk.W, pady=(0, 10))

        # Manual controls for testing
        test_frame = ttk.Frame(info_frame)
        test_frame.grid(row=6, column=0, sticky=tk.W, pady=(10, 0))

        ttk.Label(test_frame, text="Test Controls:", font=('TkDefaultFont', 9, 'bold')).pack(anchor=tk.W)

        btn_frame = ttk.Frame(test_frame)
        btn_frame.pack(anchor=tk.W, pady=(5, 0))
```

```
        ttk.Button(btn_frame, text="Refill", command=self.simulate_refill, width=8).pack(side=tk.LEFT, padx=(0, 5))
        ttk.Button(btn_frame, text="Pour", command=self.simulate_pour, width=8).pack(side=tk.LEFT)

        # Draw initial hopper
        self.draw_hopper()

    def draw_hopper(self):
        """Draw the hopper with current fill level"""
        self.hopper_canvas.delete("all")

        # Hopper dimensions
        canvas_width = 200
        canvas_height = 150
        hopper_width = 120
        hopper_height = 120
        hopper_x = (canvas_width - hopper_width) // 2
        hopper_y = 20

        # Calculate fill level
        current_weight = self.current_hopper_weight.get()
        max_weight = self.max_hopper_weight.get()

        if max_weight > 0:
            fill_ratio = min(current_weight / max_weight, 1.0)
        else:
            fill_ratio = 0

        # Draw hopper outline (trapezoid shape)
        hopper_top_width = hopper_width
        hopper_bottom_width = 60

        # Hopper coordinates
        x1 = hopper_x
        y1 = hopper_y
        x2 = hopper_x + hopper_top_width
        y2 = hopper_y
        x3 = hopper_x + hopper_top_width - (hopper_top_width - hopper_bottom_width) // 2
        y3 = hopper_y + hopper_height
        x4 = hopper_x + (hopper_top_width - hopper_bottom_width) // 2
        y4 = hopper_y + hopper_height

        # Draw hopper outline
        self.hopper_canvas.create_polygon(x1, y1, x2, y2, x3, y3, x4, y4,
                                          outline='black', fill='lightgray', width=2)

        # Draw pellets (fill level)
        if fill_ratio > 0:
            fill_height = hopper_height * fill_ratio
            fill_y_start = hopper_y + hopper_height - fill_height

            # Calculate width at fill level
            height_ratio = fill_height / hopper_height
            fill_width_top = hopper_bottom_width + (hopper_top_width - hopper_bottom_width) * (1 - height_ratio)

            # Fill coordinates
            fx1 = hopper_x + (hopper_top_width - fill_width_top) // 2
            fy1 = fill_y_start
            fx2 = fx1 + fill_width_top
            fy2 = fill_y_start
            fx3 = x3
            fy3 = y3
            fx4 = x4
            fy4 = y4

            # Choose fill color based on level
            if fill_ratio > 0.5:
                fill_color = 'lightgreen'
            elif fill_ratio > 0.2:
                fill_color = 'yellow'
            else:
                fill_color = 'lightcoral'

            self.hopper_canvas.create_polygon(fx1, fy1, fx2, fy2, fx3, fy3, fx4, fy4,
                                              fill=fill_color, outline='darkgreen', width=1)

            # Draw pellet texture
            self.draw_pellet_texture(fx1, fy1, fx2, fy2, fx3, fy3, fx4, fy4)

        # Draw spout
        spout_width = 20
        spout_height = 15
        spout_x = hopper_x + (hopper_top_width - spout_width) // 2 + (hopper_top_width - hopper_bottom_width) // 2
        spout_y = hopper_y + hopper_height

        self.hopper_canvas.create_rectangle(spout_x, spout_y, spout_x + spout_width, spout_y + spout_height,
                                            fill='gray', outline='black', width=2)

        # Update labels
        self.weight_remaining_label.config(text=f"{current_weight:.1f} g")

        fill_percent = int(fill_ratio * 100)
        self.fill_percentage_label.config(text=f"{fill_percent}%")

        # Update fill percentage label color
        if fill_percent > 50:
            self.fill_percentage_label.config(foreground="green")
        elif fill_percent > 20:
            self.fill_percentage_label.config(foreground="orange")
        else:
            self.fill_percentage_label.config(foreground="red")

    def draw_pellet_texture(self, x1, y1, x2, y2, x3, y3, x4, y4):
```

```
        """Draw small circles to represent pellets"""
        # Simple pellet representation with small circles
        pellet_size = 4
        spacing = 8

        # Calculate bounds
        min_x = min(x1, x4)
        max_x = max(x2, x3)
        min_y = y1
        max_y = max(y3, y4)

        for y in range(int(min_y + pellet_size), int(max_y), spacing):
            # Calculate width at this height
            height_ratio = (y - min_y) / (max_y - min_y)
            width_at_y = min_x + (max_x - min_x) * height_ratio
            left_bound = min_x + (width_at_y - min_x) * height_ratio / 2
            right_bound = max_x - (width_at_y - min_x) * height_ratio / 2

            for x in range(int(left_bound), int(right_bound), spacing):
                if x + pellet_size < right_bound:
                    self.hopper_canvas.create_oval(x, y, x + pellet_size, y + pellet_size,
                                                   fill='darkgreen', outline='')

    def simulate_pour(self):
        """Simulate a pour operation for testing"""
        try:
            pour_amount = float(self.pour_weight.get()) if self.pour_weight.get() else 5.0
            current = self.current_hopper_weight.get()
            new_weight = max(0, current - pour_amount)
            self.current_hopper_weight.set(new_weight)
            self.draw_hopper()

            # Reset pour timer properly
            if self.time_between.get().isdigit() and int(self.time_between.get()) > 0:
                self.last_pour_time = time.time()

            self.log_message(f"Simulated pour: {pour_amount}g (Remaining: {new_weight:.1f}g)")

            # Check if refill is needed
            max_weight = self.max_hopper_weight.get()
            if new_weight / max_weight < 0.1:  # Less than 10% remaining
                self.refill_indicator.config(text="YES", foreground="red")
                self.log_message("Refill needed!")

        except ValueError:
            self.log_message("Error: Invalid pour weight value")

    def simulate_refill(self):
        """Simulate refilling the hopper"""
        max_weight = self.max_hopper_weight.get()
        self.current_hopper_weight.set(max_weight)
        self.refill_indicator.config(text="NO", foreground="green")
        self.draw_hopper()
        self.log_message(f"Hopper refilled to {max_weight:.1f}g")

    def set_hopper_capacity(self):
        """Set the maximum hopper capacity"""
        try:
            new_capacity = float(self.hopper_weight.get())
            self.max_hopper_weight.set(new_capacity)

            # If current weight exceeds new capacity, adjust it
            if self.current_hopper_weight.get() > new_capacity:
                self.current_hopper_weight.set(new_capacity)

            self.draw_hopper()
            self.send_command(f"hopper {self.hopper_weight.get()}")
            self.log_message(f"Hopper capacity set to {new_capacity:.1f}g")
        except ValueError:
            messagebox.showerror("Invalid Value", "Please enter a valid number for hopper capacity")

    def update_countdown_timer(self):
        """Update the countdown timer for next pour"""
        try:
            if (self.last_pour_time and
                self.time_between.get().isdigit() and
                int(self.time_between.get()) > 0 and
                self.running_indicator.cget('text') == "YES"):

                time_between = int(self.time_between.get())
                elapsed = time.time() - self.last_pour_time
                remaining = max(0, time_between - int(elapsed))

                if remaining > 0:
                    mins, secs = divmod(remaining, 60)
                    if mins > 0:
                        self.next_pour_label.config(text=f"{mins}m {secs}s", foreground="orange")
                    else:
                        if secs <= 5:
                            self.next_pour_label.config(text=f"{secs}s", foreground="red")
                        else:
                            self.next_pour_label.config(text=f"{secs}s", foreground="orange")
                else:
                    self.next_pour_label.config(text="Ready", foreground="green")
                    # Note: Timer stays at "Ready" until next pour resets it
            else:
                if self.running_indicator.cget('text') == "YES":
                    self.next_pour_label.config(text="Running", foreground="blue")
                else:
                    self.next_pour_label.config(text="Stopped", foreground="gray")
        except Exception as e:
            self.next_pour_label.config(text="Error", foreground="red")
```

```
        # Schedule next update
        self.root.after(1000, self.update_countdown_timer)

    def get_serial_ports(self):
        """Get a list of available serial ports"""
        import serial.tools.list_ports
        ports = serial.tools.list_ports.comports()
        return [port.device for port in ports] if ports else ["COM3", "COM4", "COM5"]

    def toggle_connection(self):
        if not self.connected:
            self.connect_to_arduino()
        else:
            self.disconnect_from_arduino()

    def connect_to_arduino(self):
        try:
            self.serial_port = serial.Serial(self.port_var.get(), 9600, timeout=1)
            time.sleep(2)  # Wait for Arduino to reset
            self.connected = True
            self.connect_btn.config(text="Disconnect")
            self.status_label.config(text="Connected", foreground="green")

            # Start reading thread
            self.running = True
            self.reading_thread = threading.Thread(target=self.read_serial_data)
            self.reading_thread.daemon = True
            self.reading_thread.start()

            self.log_message("Connected to Arduino")

        except Exception as e:
            messagebox.showerror("Connection Error", f"Failed to connect: {str(e)}")

    def disconnect_from_arduino(self):
        self.running = False
        self.connected = False

        if self.serial_port and self.serial_port.is_open:
            self.serial_port.close()

        self.connect_btn.config(text="Connect")
        self.status_label.config(text="Disconnected", foreground="red")
        self.log_message("Disconnected from Arduino")

    def read_serial_data(self):
        buffer = ""
        while self.running and self.connected:
            try:
                if self.serial_port and self.serial_port.in_waiting:
                    # Read available data
                    data = self.serial_port.read(self.serial_port.in_waiting).decode('utf-8', errors='ignore')
                    buffer += data

                    # Process complete lines
                    while '\n' in buffer:
                        line, buffer = buffer.split('\n', 1)
                        line = line.strip()
                        if line:
                            self.root.after(0, lambda msg=line: self.log_message(f"Arduino: {msg}"))
                            self.root.after(0, lambda msg=line: self.parse_arduino_response(msg))

                time.sleep(0.05)  # Reduced sleep time for better responsiveness
            except Exception as e:
                self.root.after(0, lambda err=str(e): self.log_message(f"Read error: {err}"))
                break

    def parse_arduino_response(self, response):
        """Parse Arduino responses to update GUI status and hopper display"""
        # Log what we're parsing for debugging
        self.log_message(f"Parsing: {response}")

        if "System started" in response:
            self.running_indicator.config(text="YES", foreground="green")
            # Reset pour timer when system starts
            if self.time_between.get().isdigit() and int(self.time_between.get()) > 0:
                self.last_pour_time = time.time()
                self.log_message("Pour timer started")
        elif "System stopped" in response:
            self.running_indicator.config(text="NO", foreground="red")
            # Stop pour timer when system stops
            self.last_pour_time = None
            self.log_message("Pour timer stopped")
        elif "Refill needed" in response or "Refill Required" in response:
            self.refill_indicator.config(text="YES", foreground="red")
        elif "Refill:N" in response:
            self.refill_indicator.config(text="NO", foreground="green")

        # Look for various pour-related messages
        elif any(keyword in response.lower() for keyword in ["poured:", "dispensed:", "pour complete", "manual pour"]):
            self.handle_pour_event(response)

        # Look for weight readings
        elif any(keyword in response.lower() for keyword in ["current weight:", "weight:", "remaining:"]):
            self.handle_weight_update(response)

        # Look for timing information
        elif any(keyword in response.lower() for keyword in ["next pour in", "time remaining", "countdown"]):
            self.handle_timing_update(response)

    def handle_pour_event(self, response):
        """Handle pour-related messages from Arduino"""
        try:
```

```
            # Try to extract pour amount from various message formats
            pour_amount = None
            response_lower = response.lower()

            # Try different patterns
            if "poured:" in response_lower:
                pour_amount = float(response.split("Poured:")[1].split("g")[0].strip())
            elif "dispensed:" in response_lower:
                pour_amount = float(response.split("Dispensed:")[1].split("g")[0].strip())
            elif "pour complete" in response_lower:
                # Use the set pour weight if no amount specified
                pour_amount = float(self.pour_weight.get()) if self.pour_weight.get() else 0
            elif "manual pour" in response_lower:
                pour_amount = float(self.pour_weight.get()) if self.pour_weight.get() else 0

            if pour_amount is not None and pour_amount > 0:
                # Update hopper weight
                current = self.current_hopper_weight.get()
                new_weight = max(0, current - pour_amount)
                self.current_hopper_weight.set(new_weight)
                self.draw_hopper()

                # Reset pour timer
                if self.time_between.get().isdigit() and int(self.time_between.get()) > 0:
                    self.last_pour_time = time.time()
                    self.log_message(f"Pour detected: {pour_amount}g. Timer reset. Remaining: {new_weight:.1f}g")

        except (ValueError, IndexError, AttributeError) as e:
            self.log_message(f"Error parsing pour amount: {e}")

    def handle_weight_update(self, response):
        """Handle weight update messages from Arduino"""
        try:
            # Try to extract weight from various message formats
            weight = None
            response_lower = response.lower()

            if "current weight:" in response_lower:
                weight = float(response.split("Current weight:")[1].split("g")[0].strip())
            elif "weight:" in response_lower:
                weight = float(response.split("weight:")[1].split("g")[0].strip())
            elif "remaining:" in response_lower:
                weight = float(response.split("remaining:")[1].split("g")[0].strip())

            if weight is not None and weight >= 0:
                self.current_hopper_weight.set(weight)
                self.draw_hopper()
                self.log_message(f"Weight updated to: {weight:.1f}g")

        except (ValueError, IndexError, AttributeError) as e:
            self.log_message(f"Error parsing weight: {e}")

    def handle_timing_update(self, response):
        """Handle timing-related messages from Arduino"""
        try:
            # This could be used if Arduino sends timing info
            # For now, we'll rely on our own timer
            pass
        except Exception as e:
            self.log_message(f"Error parsing timing: {e}")

    def send_command(self, command):
        if not self.connected or not self.serial_port:
            messagebox.showwarning("Not Connected", "Please connect to Arduino first")
            return

        # Use threading to prevent GUI blocking
        def send_in_thread():
            try:
                self.serial_port.write(f"{command}\n".encode())
                self.serial_port.flush()  # Ensure data is sent immediately
                self.root.after(0, lambda: self.log_message(f"Sent: {command}"))
            except Exception as e:
                self.root.after(0, lambda: messagebox.showerror("Send Error", f"Failed to send command: {str(e)}"))

        # Send command in separate thread to avoid blocking
        send_thread = threading.Thread(target=send_in_thread)
        send_thread.daemon = True
        send_thread.start()

    def start_system(self):
        self.start_btn.config(state='disabled')  # Prevent multiple clicks
        self.send_command("start")

        # Start pour timer if system starts
        if self.time_between.get().isdigit():
            self.last_pour_time = time.time()

        self.root.after(1000, lambda: self.start_btn.config(state='normal'))  # Re-enable after 1 second

    def stop_system(self):
        self.stop_btn.config(state='disabled')  # Prevent multiple clicks
        self.send_command("stop")

        # Stop pour timer
        self.last_pour_time = None

        self.root.after(1000, lambda: self.stop_btn.config(state='normal'))  # Re-enable after 1 second

    def manual_pour(self):
        self.manual_btn.config(state='disabled')
        self.send_command("manual")
        self.root.after(1000, lambda: self.manual_btn.config(state='normal'))
```

```
    def calibrate(self):
        self.calibrate_btn.config(state='disabled')
        self.send_command("calibrate")
        self.root.after(2000, lambda: self.calibrate_btn.config(state='normal'))  # Longer delay for calibration

    def get_status(self):
        self.status_btn.config(state='disabled')
        self.send_command("status")
        self.root.after(500, lambda: self.status_btn.config(state='normal'))

    def alarm_on(self):
        self.alarm_on_btn.config(state='disabled')
        self.send_command("alarm_on")
        self.root.after(500, lambda: self.alarm_on_btn.config(state='normal'))

    def alarm_off(self):
        self.alarm_off_btn.config(state='disabled')
        self.send_command("alarm_off")
        self.root.after(500, lambda: self.alarm_off_btn.config(state='normal'))

    def log_message(self, message):
        timestamp = datetime.now().strftime("%H:%M:%S")
        self.output_text.insert(tk.END, f"[{timestamp}] {message}\n")
        self.output_text.see(tk.END)

    def clear_output(self):
        self.output_text.delete(1.0, tk.END)

    def update_settings_display(self):
        settings = f"""Calibration: {self.cal_weight.get()} g
Pour Weight: {self.pour_weight.get()} g
Time Between: {self.time_between.get()} s
Hopper Capacity: {self.hopper_weight.get()} g

System Running: {self.running_indicator.cget('text')}
Refill Needed: {self.refill_indicator.cget('text')}"""

        self.settings_text.delete(1.0, tk.END)
        self.settings_text.insert(1.0, settings)

        # Schedule next update
        self.root.after(2000, self.update_settings_display)

    def on_closing(self):
        self.running = False
        if self.connected:
            self.disconnect_from_arduino()
        self.root.destroy()

def main():
    root = tk.Tk()
    app = PelletDispenserGUI(root)

    # Handle window closing
    root.protocol("WM_DELETE_WINDOW", app.on_closing)

    root.mainloop()

if __name__ == "__main__":
    main()
```