Arduino-Python Serial LED Control

ArduinoLED...roller.ino
Unknown

LED_Controller
H

Using these two files (Arduino Sketch and Header), create the Python part using the following instructions below:

Objectives:
- Understand and implement Arduino Serial Connection
- Utilize Python as a tool for implementing serial connection
- Create a simple circuit that can be controlled using Serial connection using Arduino and Python

Instructions:
Using the laboratory guide, implement the following requirement
1. Use the following components:
- Arduino MCU
- 3 LEDs (Red, Green, Blue recommended)
- Usual components (wires, breadboard, resistors, laptop with Python and pyserial installed)

2. Use the following pins:
- Red - 8 Green - 9 Blue - 10

3. In your Arduino, create the sketch program that will allow serial input. The following actions are expected for the inputs
- R/r = it should toggle only the red LED on/off
- G/g = it should toggle only the green LED on/off
- B/b = it should toggle only the blue LED on/off
- A/a = it should turn all LEDs on
- O/o = it should turn all LEDs off
- all other inputs should return an error

4. Using Python, create a non-terminating script that will show the following choices, and be able to execute the same logic as in Arduino.

[R] Red ON/OFF
[G] Green ON/OFF
[B] Blue ON/OFF
[A] All ON
[O] All OFF
[X] Exit

5. When X/x is inputted, it should terminate the Python application from running.

6. All inputs should be case insensitive.

Nice to haves:
- In Python, clear the screen for every input made, then reprint the choices (5 pts)

- In Arduino, place all common functions inside a header file.

Based on the Arduino sketch and header file you provided, here is the corresponding Python script to control your circuit.

This script implements the requirements: it utilizes `pyserial` for communication, handles the menu interface, clears the screen (Nice to have #1), and loops until the exit command is received.

Prerequisites

Before running the script, ensure you have the `pyserial` library installed. Open your terminal or command prompt and run:

Bash

```
pip install pyserial
```

The Python Script (`led_controller.py`)

You will need to update the `SERIAL_PORT` variable (e.g., `COM3` on Windows or `/dev/ttyUSB0` on Mac/Linux) to match the port your Arduino is connected to.

Python

```
import serial
import time
import os
import sys

# ========================================
# CONFIGURATION
# ========================================
# REPLACE 'COM3' with your actual Arduino Port
# Windows: 'COM3', 'COM4', etc.
# Mac/Linux: '/dev/tty.usbmodem...' or '/dev/ttyUSB0'
SERIAL_PORT = 'COM3'
BAUD_RATE = 9600

def clear_screen():
    """Clears the console screen based on the OS."""
    # 'cls' for Windows, 'clear' for Mac/Linux
    os.system('cls' if os.name == 'nt' else 'clear')

def main():
    # 1. Setup Serial Connection
    try:
        arduino = serial.Serial(port=SERIAL_PORT, baudrate=BAUD_RATE, timeout=1)
        # Wait for Arduino to reset after connection opens
        print(f"Connecting to {SERIAL_PORT}...")
        time.sleep(2)
        print("Connection established!")
        time.sleep(1)
    except serial.SerialException:
        print(f"Error: Could not connect to {SERIAL_PORT}.")
        print("Check your cable and ensure the correct port is specified in the code.")
        sys.exit()

    while True:
        # 2. Clear Screen and Print Menu (Nice to Have)
        clear_screen()
        print("=== Arduino LED Controller ===")
        print("[R] Red ON/OFF")
```

```
        print("[G] Green ON/OFF")
        print("[B] Blue ON/OFF")
        print("[A] All ON")
        print("[O] All OFF")
        print("[X] Exit")
        print("============================")

        # 3. Get User Input (Case Insensitive)
        choice = input("Enter selection: ").strip().upper()

        # 4. Handle Exit
        if choice == 'X':
            print("Terminating application...")
            arduino.close()
            break

        # 5. Handle Valid Commands
        if choice in ['R', 'G', 'B', 'A', 'O']:
            # Send the character + newline because Arduino uses readStringUntil('\n')
            command = choice + '\n'
            arduino.write(command.encode('utf-8'))

            # Read feedback from Arduino (Optional, verifies the hardware worked)
            time.sleep(0.1) # Give Arduino a moment to process
            if arduino.in_waiting > 0:
                response = arduino.readline().decode('utf-8').strip()
                print(f"\nArduino response: {response}")

            # Pause briefly so the user can read the response before the screen clears
            input("\nPress Enter to continue...")

        else:
            print("\nInvalid input! Please select from the menu.")
            time.sleep(1.5)

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

Circuit Wiring Guide

To make this work with your provided `LED_Controller.h`, wire your breadboard as follows:

1. **Ground:** Connect the Arduino **GND** pin to the blue (-) rail of the breadboard.
2. **Red LED:**
   - **Anode (+, long leg):** Connect to Arduino **Pin 8**.
   - **Cathode (-, short leg):** Connect to a 220Ω or 330Ω resistor, then to **GND**.
3. **Green LED:**
   - **Anode (+):** Connect to Arduino **Pin 9**.
   - **Cathode (-):** Connect to a resistor, then to **GND**.
4. **Blue LED:**
   - **Anode (+):** Connect to Arduino **Pin 10**.
   - **Cathode (-):** Connect to a resistor, then to **GND**.

How It Works

1. **Arduino Side:** Your sketch waits for `Serial.readStringUntil('\n')`. It trims whitespace, converts the character to uppercase, and calls the functions in your header file (`toggleRed`, `allOn`, etc.).
2. **Python Side:**
   - The script opens the COM port.

- It enters a `while True` loop to keep the program running.
- It captures your input (e.g., 'r'), converts it to 'R', and sends it over the USB cable.
- It appends a `\n` (newline) when sending, which is crucial because your Arduino code uses `readStringUntil('\n')`.
- It reads the response (e.g., "Red LED toggled.") printed by the Arduino to confirm execution.

Why does it need to try-catch the serial port?
Additionally what is the function of arduino.close(), arduino.write() in this code?

Here is an explanation of why those specific functions and blocks are used in your Python script.

1. Why use a `try-catch` (try-except) block for the Serial Port?

In programming, communicating with external hardware (like an Arduino) is considered **"risky" code**. Unlike adding two numbers ($1 + 1$), connecting to a port depends on physical factors outside your code's control.

The `try-except` block is a safety net. Without it, if something goes wrong, your program will crash instantly with a confusing "Traceback" error message.

Common reasons the connection fails:

- **Wrong Port:** You wrote `COM3`, but the Arduino is plugged into `COM4`.
- **Port Busy:** The **Arduino IDE Serial Monitor** is still open. (This is the most common error! Only one program can control the serial port at a time).
- **Unplugged:** The USB cable is disconnected or loose.

With the `try-except` block: Instead of crashing, the program catches the specific `serial.SerialException`, prints a helpful message ("Check your cable..."), and closes neatly. This is a key principle in **HCI (Human-Computer Interaction)**—handling errors gracefully rather than frightening the user.

2. `arduino.write()`

This function is the "mouth" of your Python script. It is responsible for actually sending the data over the USB cable to the Arduino.

- **How it works:** It takes a string of text, converts it into raw bytes (using `.encode('utf-8')`), and pushes it down the wire.
- **Relevance to your code:** In your Arduino sketch, you used this line:

  `String inputString = Serial.readStringUntil('\n');`

  This means your Arduino is waiting for a **New Line** character (`\n`) to know the command is finished. Therefore, in Python, we cannot just send `'R'`. We must send `'R\n'`.

  `arduino.write(command.encode('utf-8'))` does exactly that—it sends the character plus the newline you appended so the Arduino knows to process it.

3. `arduino.close()`

This function is the "cleanup crew." It formally disconnects the Python script from the USB port.

Why is it important? Operating Systems (Windows/Mac/Linux) place a **lock** on a serial port when a program opens it.

- **If you DO use `close()`:** The OS releases the lock immediately. You can then instantly upload new code from the Arduino IDE or restart your Python script without issues.
- **If you DO NOT use `close()`:** If you force-quit the script (or if it crashes without the `try-except` block), the OS might think the port is **still in use**.
  - When you try to run the script again, you will get an **"Access Denied"** or **"Resource Busy"** error.
  - You would then have to unplug the USB cable or restart your computer to free the port.

Summary Table

Code	Purpose	Real-world Analogy
`try-except`	Error Handling	Checking if the door is locked before trying to barge in, so you don't hurt your shoulder.
`arduino.write()`	Data Transmission	Speaking into a walkie-talkie to give a command.
`arduino.close()`	Resource Management	Hanging up the phone so the line is free for the next call.

What is the difference of sys.exit and arduino.close()?

The main difference is what they target: `arduino.close()` **targets the hardware connection**, while `sys.exit()` **targets the software application.**

Here is the breakdown of why you need both and how they differ:

1. `arduino.close()` (The Hardware Manager)

- **What it does:** It tells your Operating System (Windows/Mac/Linux) to release the specific USB/COM port that your Python script is currently holding.
- **Why it matters:** Serial ports are **exclusive resources**. If your Python script grabs `COM3`, no other program (not even the Arduino IDE) can touch `COM3` until your script lets go.
- **If you forget it:** If you stop your Python script without running this line, the OS might think `COM3` is still in use. When you try to run your script again, you will get an `Access Denied` or `Resource Busy` error, forcing you to unplug the USB cable or restart your PC.

2. `sys.exit()` (The Program Terminator)

- **What it does:** It stops the Python script entirely. It raises a specific exception (`SystemExit`) that kills the running process.
- **Why it matters:** In your code, you are running a `while True:` loop (an infinite loop). Without `sys.exit()` (or a `break` statement that leads to the end of the file), the script would run forever, constantly asking for input or printing to the console.
- **If you forget it:** Your program will stay stuck in the loop, consuming CPU memory and requiring you to "Force Quit" the terminal window.

The "Store Closing" Analogy

Think of your Python script as a **Store** and the Serial Connection as the **Front Door**.

1. `arduino.close()` **is locking the front door.** You do this so no one else can come in, and you formally secure the building. If you don't do this, the door stays wide open, and the lock might jam when you try to use it tomorrow.
2. `sys.exit()` **is the employees going home.** This is the act of actually leaving the building and ending the workday.

The Correct Order

You must always **close the connection first**, then **exit the program**.

Python

```
# CORRECT
arduino.close()  # 1. Release the hardware resource (Lock the door)
sys.exit()       # 2. Kill the program (Go home)
```

If you reverse them, `sys.exit()` will kill the program immediately, and the line `arduino.close()` will never be reached, potentially leaving your port locked!

Next Step

Would you like me to show you how to wrap this in a `try...finally` block? This ensures that `arduino.close()` runs **automatically** even if your program crashes or you force-quit it with `Ctrl+C`.

Can I put sys.exit() when the user choose X:
"When X/x is inputted, it should terminate the Python application from running."

Yes, you absolutely can use `sys.exit()` to satisfy that requirement. In fact, since the instruction explicitly says "terminate the Python application," `sys.exit()` is a very literal and correct way to handle it.

However, you must be careful to **close the connection first**.

Here is how you would modify the code to use `sys.exit()` instead of `break`:

Python

```
import sys  # Make sure you import sys at the top

# ... inside your loop ...

if choice == 'X':
    print("Terminating application...")

    # 1. ALWAYS Close the port first
    arduino.close()

    # 2. Then kill the application
    sys.exit()
```

The Difference: `sys.exit()` **vs.** `break`

While both will stop your script in this specific scenario, they behave differently if you add more code later:

1. `break`: This just "breaks out" of the loop. If you had code **below** the `while` loop (like a "Goodbye" message or a final log save), that code **would still run**.

2. `sys.exit()`: This is the "Nuclear Option." It kills the program immediately at that specific line. Any code below the loop will **never** be executed.

Which one should you use?

For your specific assignment, `sys.exit()` is perfect because the instructions ask to "terminate the application" immediately. Just remember the golden rule: **Close the port, then exit.**