Day 5 Session 4:
pySerial and PyUSB

This notebook focuses on **UART (Universal Asynchronous Receiver-Transmitter) serial communication** using the pyserial Python library, and introduces **Direct USB Communication** using PyUSB. These are fundamental methods for direct, low-level data exchange between a computer and various hardware devices like microcontrollers, sensors, and specialized instruments. These are crucial skills for embedded systems engineers and anyone interfacing with hardware.

1. Basics of Serial (COM) Communication

Imagine sending data one bit at a time, in a single file line, from one device to another. That's serial communication!

- **Serial Communication:** Data bits are sent sequentially over a single wire (or a pair of wires for transmit/receive). This is different from parallel communication, where multiple bits are sent simultaneously over multiple wires.

- **UART:** This is the hardware part (or protocol) inside devices that handles this asynchronous serial communication (meaning no shared clock signal between sender and receiver).

- **COM Port (Windows) / tty device (Linux/macOS):** This is how your operating system sees a serial port. It could be a physical port on an older computer, or more commonly, a virtual serial port created by a USB-to-Serial converter or a microcontroller development board plugged into USB.

Key Serial Parameters (for both sender and receiver to agree on):

For devices to talk to each other serially, they must agree on these settings:

- **Baud Rate:** The speed of data transfer, measured in bits per second (bps). Common rates: 9600, 19200, 57600, 115200. Both devices *must* use the same baud rate.
- **Data Bits:** Number of data bits in each character (usually 8).

- **Parity:** An optional error-checking bit (None, Even, Odd). Usually None.
- **Stop Bits:** Bits that mark the end of a character (usually 1).
- **Timeout:** How long to wait for data before giving up (important for reading).

Why Use Serial Communication (UART) with Python?

- **Microcontroller Interface:** Talk directly to embedded systems for control, data logging, and firmware updates.
- **Legacy Instruments:** Many older scientific and industrial instruments use RS-232 serial.
- **Simple & Direct:** A straightforward way to communicate with hardware without complex network stacks.
- **Debugging:** Monitor output from embedded devices.

2. pyserial for UART

The pyserial library is the standard Python module for serial port access. It works across Windows, Linux, and macOS.

Installation: Open your terminal/command prompt and run:

```Shell
pip install pyserial
```

2.1 Listing Available Serial Ports (serial.tools.list_ports.comports())

Before connecting, you need to know which serial ports are available on your system.

serial.tools.list_ports.comports(): How to Find Serial Ports

- **Purpose:** Discovers and lists all detected serial (COM) ports on your computer.
- **Syntax:** ports = serial.tools.list_ports.comports()
- **Output:** Returns a list of UsbPortInfo objects. Each object has a device (port name), description (human-readable), and hwid (hardware ID).

Python

```python
# ------------------------------------------------------------
# Serial Port Detection using pyserial
# This script lists all available serial (COM) ports on your system.
# Works on Windows (COMx), Linux (/dev/ttyUSBx), and macOS (/dev/cu.x).
# ------------------------------------------------------------

import serial.tools.list_ports  # Import the submodule to list serial ports

print("--- Listing Available Serial Ports ---")

try:
    # Get a list of all serial ports currently connected
    available_ports = serial.tools.list_ports.comports()

    if available_ports:
        print("✅ Found the following serial ports:")
        for port in available_ports:
            print(f" - Device: {port.device}")            # The port name (e.g., COM3, /dev/ttyUSB0)
            print(f"   Description: {port.description}") # A human-readable description of the port
            print(f"   Hardware ID: {port.hwid}")         # The unique hardware ID (useful for debugging)
            print("-" * 40)

        print("\n💡 Tip: In your code, use one of the listed 'Device' names instead of a placeholder like 'COM3'.")
    else:
        print("⚠️ No serial ports found.")
        print("   This is normal if no USB/Serial devices are connected (like Arduino, USB-to-Serial adapters).")
        print("   You can still write and test code using a dummy port name like '/dev/ttyUSB0' or 'COM4'.")
except Exception as e:
    print(f"❌ Error listing ports: {e}")
    print("Make sure pyserial is installed and you have permission to access serial devices.")
```

2.2 Setting Up the Serial Port (serial.Serial())

To communicate, you create a Serial object, providing the port name and the agreed-upon parameters (baud rate, etc.).

serial.Serial(): How to Open a Serial Port

- **Purpose:** Opens a connection to a serial port with specified settings.
- **Syntax:** ser = serial.Serial(port, baudrate, bytesize=8, parity='N', stopbits=1, timeout=None, ...)
    - port: The name of the serial port (e.g., 'COM3' on Windows, '/dev/ttyUSB0' on Linux).
    - baudrate: Data transfer speed (e.g., 9600, 115200).
    - timeout: (Optional) How long to wait (in seconds) for a read operation to complete. None means wait forever. 0 means non-blocking. A positive number means blocking with a timeout.

Important: Always Close Connections!

Just like files, you must close your serial port connection when you're done. The with statement (as a context manager) is the best way to do this, as it guarantees the port is closed even if errors occur.

Python
```python
# ------------------------------------------------------------
# Serial Port Setup Example using PySerial
# This script opens a serial port and prints its current settings.
# ------------------------------------------------------------

import serial       # For serial communication
import time         # For introducing delays

print("\n--- Setting Up the Serial Port ---")

# 🧠 STEP 1: Replace this with the actual serial port detected earlier
# For Windows, use something like 'COM3'
# For Linux, use something like '/dev/ttyUSB0' or '/dev/ttyACM0'
DUMMY_PORT = 'COM3'  # ← 🔁 Change this to your actual port name
```

```
# 🧠 STEP 2: Set the baud rate (must match the connected device)
BAUD_RATE = 9600     # Most Arduino boards and many sensors use this

try:
    # ✅ STEP 3: Open the serial port using a context manager (auto-closes on exit)
    with serial.Serial(port=DUMMY_PORT, baudrate=BAUD_RATE, timeout=1) as ser:
        print(f"✅ Successfully opened serial port: {ser.port}")
        print(f"   Baud Rate     : {ser.baudrate}")
        print(f"   Byte Size     : {ser.bytesize}")
        print(f"   Parity        : {ser.parity}")
        print(f"   Stop Bits     : {ser.stopbits}")
        print(f"   Timeout (sec) : {ser.timeout}")
        print("🚀 Serial port is now active and ready to use.")

        # ⏳ Give the device a moment to get ready (common for Arduino)
        time.sleep(1)

        # 🔄 Add communication code here: ser.write(...) or ser.read(...)

    # ✅ STEP 4: Port automatically closes after 'with' block ends
    print(f"🛑 Serial port {DUMMY_PORT} is now closed.")

except serial.SerialException as e:
    print(f"❌ Error opening serial port {DUMMY_PORT}: {e}")
    print("🔍 This usually happens if the port doesn't exist, is already in use, or no device is connected.")
    print("💡 Tip: Run the port listing script to confirm the correct port name.")
except Exception as e:
    print(f"❌ An unexpected error occurred: {e}")
```

3. Reading and Writing Data Packets (UART)

Once the serial port is open, you can send (write) data to the device and receive (read) data from it.

3.1 Writing Data (ser.write())

Conceptual Approach:

To send data, you use the ser.write() method. Just like sockets, data must be in bytes.

ser.write(): How to Send Data

- **Purpose:** Writes data (bytes) to the serial port.
- **Syntax:** ser.write(data_bytes)
  - data_bytes: The data to send, *must be a bytes object* (e.g., b'Hello', 'Hello'.encode('utf-8')).

3.2 Reading Data (ser.read(), ser.readline())

Conceptual Approach:

To read data, you use methods like ser.read() (reads a specific number of bytes) or ser.readline() (reads until a newline character or timeout). Data received is always in bytes, so you'll decode() it to a string.

Serial Port Communication Methods

Method	Purpose
\`ser.read(size)\`	Reads \`size\` bytes from the port. Blocks until \`size\` bytes are received or a timeout occurs.
\`ser.readline()\`	Reads until a newline character (\`\n\`) is received. Blocks until a newline or a timeout occurs.
\`ser.in_waiting\`	Returns the number of bytes currently in the input buffer.

Python
```python
# ------------------------------------------------------------
# UART Communication Example (Read & Write)
# This script demonstrates how to send commands and read responses
# via a serial (UART) connection, e.g., to/from Arduino or sensor modules.
# ------------------------------------------------------------

import serial        # For serial communication
import time          # To pause/wait between operations
import random        # For simulating test values if needed

print("\n--- Reading and Writing Data Packets (UART) ---")

# ✅ STEP 1: Specify the correct port and baud rate
# ⚠️ Change this to match your actual connected port
DUMMY_PORT = 'COM3'     # Example: 'COM3' (Windows) or '/dev/ttyUSB0'
(Linux/RPi)
BAUD_RATE = 9600        # Must match the device's UART setting

try:
    # ✅ STEP 2: Open the serial port using a context manager
    with serial.Serial(port=DUMMY_PORT, baudrate=BAUD_RATE, timeout=2) as ser:
        print(f"✅ Connected to {ser.port} for Read/Write test.")
        time.sleep(1)  # Optional: Wait for device to initialize

        # 🔸 STEP 3: Send a request to get temperature
        command_to_send = "GET_TEMP\n"  # Many devices expect
newline-terminated commands
        ser.write(command_to_send.encode('utf-8'))  # Send as bytes
        print(f"📤 Sent command: '{command_to_send.strip()}'")

        # 🔹 STEP 4: Read the temperature response
        # Expects a response like: "TEMP:28.7\n"
        response_bytes = ser.readline()  # Reads until \n or timeout
        if response_bytes:
            device_response = response_bytes.decode('utf-8').strip()
            print(f"📥 Received response: '{device_response}'")

            # Parse temperature value from the response
            if device_response.startswith("TEMP:"):
                try:
                    temp_val = float(device_response.split(":")[1])
                    print(f"🌡️ Parsed Temperature: {temp_val} °C")
                except (ValueError, IndexError):
                    print("⚠️ Could not parse temperature from response.")
        else:
            print("⏳ No response received within timeout period.")

        # 🔸 STEP 5: Send a configuration value (e.g., set brightness)
        set_light_level = 150  # Value between 0 and 255
        config_command = f"SET_LIGHT:{set_light_level}\n"
        ser.write(config_command.encode('utf-8'))
        print(f"\n📤 Sent config command: '{config_command.strip()}'")

        # 🔹 STEP 6: Read acknowledgment (expecting something like 'OK' or
'ACK')
        ack_response = ser.readline().decode('utf-8').strip()
        print(f"📥 Received Acknowledgment: '{ack_response}'")

except serial.SerialException as e:
    print(f"❌ Serial communication error: {e}")
    print("💡 Check if the port exists, is free, and connected to a proper
device.")
except Exception as e:
    print(f"❌ Unexpected error: {e}")
```

4. Syncing Device Reads with File Logging (UART)

In real applications, you often receive data from a device and need to log it to a file, often with timestamps.

Conceptual Approach:

- Set up a loop to continuously read from the serial port.
- For each reading, add a timestamp and format it.
- Write the formatted log entry to a file.
- This would typically require a real device connected to DUMMY_PORT.

Python
```python
# ------------------------------------------------------------
# UART Data Logger
# This script connects to a serial device (like Arduino/Raspberry Pi),
# reads structured sensor data (e.g., temperature, pressure),
# and saves it with timestamps to a CSV-style log file.
# ------------------------------------------------------------
```

```
import serial          # For serial communication
import time            # For timing/log intervals
import datetime        # For timestamping logs
import os              # To manage files

print("\n--- Syncing Device Reads with File Logging (UART) ---")

# ✅ STEP 1: Specify serial port settings
DUMMY_PORT = 'COM3'                    # ⚠️ Replace with actual port (e.g.,
'/dev/ttyUSB0' on Linux)
BAUD_RATE = 9600             # Must match device's UART setting
LOG_FILE_NAME = "sensor_readings_log.txt"

try:
    # ✅ STEP 2: Open serial port connection
    with serial.Serial(port=DUMMY_PORT, baudrate=BAUD_RATE, timeout=1) as ser:
        print(f"📡 Connected to {ser.port}. Logging to '{LOG_FILE_NAME}'...")

        # ✅ STEP 3: Open a log file to write the readings
        with open(LOG_FILE_NAME, "w") as log_file:
            # Write header row
            log_file.write("Timestamp,Temperature_C,Pressure_kPa\n")
            print(f"📝 Log file '{LOG_FILE_NAME}' opened and ready.")

            # ✅ STEP 4: Read from UART and write to file
            start_time = time.time()
            max_log_duration = 5  # Log for 5 seconds (adjust as needed)

            while (time.time() - start_time) < max_log_duration:
                line_bytes = ser.readline()  # Wait for incoming data
                if line_bytes:
                    line_str = line_bytes.decode('utf-8').strip()
                                                            current_timestamp   =
datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")

                    # Parse example input like: "TEMP:25.5,PRES:101.2"
                    temp = "N/A"
                    pres = "N/A"
                    parts = line_str.split(',')
                    for part in parts:
                        if part.startswith("TEMP:"):
                            try:
```

```python
                                temp = float(part.split(":")[1])
                            except ValueError:
                                pass
                        elif part.startswith("PRES:"):
                            try:
                                pres = float(part.split(":")[1])
                            except ValueError:
                                pass

                    # Format row and write to file
                    log_entry = f"{current_timestamp},{temp},{pres}\n"
                    log_file.write(log_entry)
                    print(f"✅ Logged: {log_entry.strip()}")

                time.sleep(0.1)  # Prevent busy loop, allow device time to send
data

        print(f"\n✅ Logging completed. Data saved to: '{LOG_FILE_NAME}'")

except serial.SerialException as e:
    print(f"❌ Serial Port Error: {e}")
    print("💡 Make sure the correct serial port is used and device is
connected.")
except Exception as e:
    print(f"❌ Unexpected Error: {e}")
finally:
    # ✅ STEP 5: Display the logged file content (optional)
    if os.path.exists(LOG_FILE_NAME):
        print(f"\n📂 Preview of '{LOG_FILE_NAME}':")
        with open(LOG_FILE_NAME, "r") as f:
            print(f.read())
        # Optional cleanup
        # os.remove(LOG_FILE_NAME)  # Uncomment to auto-delete after view
```

5. Simulating Microcontroller Communication (Conceptual)

In a full embedded project, your Python script acts as the "host" talking to a "client" (the microcontroller). The pyserial code in this notebook (especially the logging example) is exactly what you would use to receive data from such a microcontroller.

Conceptual Microcontroller (e.g., Arduino C++ code):

C/C++
```cpp
// --- Arduino: Simulates sending sensor data every 1 second via UART ---

void setup() {
  Serial.begin(9600); // Start serial communication at 9600 baud
}

void loop() {
  // Simulate temperature between 25.0–29.9 °C
  float temperature = 25.0 + random(0, 50) / 10.0;

  // Simulate pressure between 100.0–101.9 kPa
  float pressure = 100.0 + random(0, 20) / 10.0;

  // Format and send data like: TEMP:27.3,PRES:100.6\n
  Serial.print("TEMP:");
  Serial.print(temperature);
  Serial.print(",PRES:");
  Serial.print(pressure);
  Serial.print("\n"); // Important for Python's .readline()

  delay(1000); // Wait 1 second
}
```

Python
```python
import serial
import time
import datetime
```

```
# --- Python: Reads and parses serial data from Arduino ---

PORT  =  '/dev/ttyUSB0'    #  Replace  with  your  actual  port  like  'COM3'  or
'/dev/ttyUSB0'
BAUD = 9600
TIMEOUT = 1

try:
    with serial.Serial(PORT, BAUD, timeout=TIMEOUT) as ser:
        print(f"🔌 Connected to {PORT}. Waiting for data...\n")
        time.sleep(2)  # Wait for Arduino to reset

        for _ in range(10):  # Read 10 lines as example
            line = ser.readline().decode('utf-8').strip()

            if line:
                print(f"📥 Raw: '{line}'")
                # Example: TEMP:27.3,PRES:100.6
                parts = line.split(',')
                data = {}
                for part in parts:
                    key_val = part.split(':')
                    if len(key_val) == 2:
                        key, val = key_val
                        data[key.strip()] = float(val.strip())

                # Output parsed result
                timestamp = datetime.datetime.now().strftime("%H:%M:%S")
                    print(f"🕒 {timestamp} | 🌡️ Temp: {data.get('TEMP')}°C | ⬇️
Pressure: {data.get('PRES')} kPa\n")
            else:
                print("⚠️ No data received.")

            time.sleep(1)  # Match Arduino's sending interval

except serial.SerialException as e:
    print(f"❌ Serial Error: {e}")
except Exception as e:
    print(f"❌ Unexpected Error: {e}")
```

6. Direct USB Communication with PyUSB

While pyserial handles USB devices that appear as serial (COM) ports, some USB devices have their own custom communication protocols and don't create a virtual serial port. For these, you need to talk directly to the USB device using a lower-level library like PyUSB.

Conceptual Approach:

PyUSB allows you to access the raw "endpoints" (channels) of a USB device for reading and writing. This is much more complex than pyserial because you need to understand the device's specific USB protocol (Vendor ID, Product ID, endpoint addresses, data formats).

Prerequisites for PyUSB:

1. **PyUSB library**: Install it using pip install pyusb.
2. **libusb**: This is a low-level C library that PyUSB uses to talk to the operating system's USB stack. You need to install libusb on your system.
    - Windows: You might need Zadig to install WinUSB drivers for your specific device.
    - Linux/macOS: Often available through package managers (apt install libusb-1.0-0-dev on Debian/Ubuntu).
3. **Device Drivers/Permissions**: Sometimes, you need to detach the default operating system driver from the device or set specific user permissions to access it.

usb.core.find(): How to Find a USB Device

- Purpose: Searches for a USB device based on its Vendor ID (vid) and Product ID (pid).
- Syntax: dev = usb.core.find(idVendor=0xXXXX, idProduct=0xYYYY)

Basic Workflow with PyUSB:

1. Find the device: Use usb.core.find().
2. Set configuration: Choose which operating mode the device should use (dev.set_configuration()).
3. Claim interface (if needed): Take control of a communication interface

   (usb.util.claim_interface()).
4. Find endpoints: Identify the input (read) and output (write) communication channels.
5. Read/Write: Send bytes to output endpoints (dev.write()) and read bytes from input endpoints (dev.read()).
6. Release/Close: Release interface and detach kernel driver (usb.util.release_interface(), usb.util.detach_kernel_driver()).

Code Example (Conceptual for a Generic USB Device):

```python
import usb.core      # Core functions for finding/configuring USB devices
import usb.util      # Utility functions (detach, release, endpoint matching)
import time          # For delays and timing

print("\n🔌 --- Direct USB Communication with PyUSB ---")

# Step 1: Replace these with your device's actual Vendor ID and Product ID
# Find these via:
#   - Windows: Device Manager → Details tab → Hardware Ids
#   - Linux/macOS: `lsusb`
MY_VENDOR_ID = 0x1234  # Example: Replace with your real Vendor ID
MY_PRODUCT_ID = 0x5678 # Example: Replace with your real Product ID

try:
    # Step 2: Find the USB device
    device = usb.core.find(idVendor=MY_VENDOR_ID, idProduct=MY_PRODUCT_ID)
    if device is None:
        raise ValueError(f"USB Device with VID=0x{MY_VENDOR_ID:X},
PID=0x{MY_PRODUCT_ID:X} not found.")

    print(f"✅ Found USB device: {device}")
    print(f"🔢 Serial Number: {device.serial_number}")

    # Step 3: Detach any existing OS kernel driver (Linux/macOS only)
    if device.is_kernel_driver_active(0):
        try:
            device.detach_kernel_driver(0)
            print("🔓 Detached kernel driver from interface 0.")
        except usb.core.USBError as e:
            print(f"⚠️ Could not detach kernel driver: {e}")
        print("Try running with `sudo` or set udev rules (Linux).")

    # Step 4: Set the device configuration
    device.set_configuration()
    print("🔧 Device configuration activated.")

    # Step 5: Get configuration and interface
    config = device.get_active_configuration()
    interface = config[(0, 0)]  # Most devices use interface 0, alt setting 0

    # Step 6: Find endpoints
    ep_out = usb.util.find_descriptor(
        interface,
        custom_match=lambda e: usb.util.endpoint_direction(e.bEndpointAddress)
== usb.util.ENDPOINT_OUT
    )
    ep_in = usb.util.find_descriptor(
        interface,
        custom_match=lambda e: usb.util.endpoint_direction(e.bEndpointAddress)
== usb.util.ENDPOINT_IN
    )

    if not ep_out or not ep_in:
        raise ValueError("❌ Could not find both IN and OUT endpoints. Check
device documentation.")

    print(f"➡️  OUT Endpoint Address: 0x{ep_out.bEndpointAddress:X}")
    print(f"⬅️  IN Endpoint Address: 0x{ep_in.bEndpointAddress:X}")

    # Step 7: Send a command to the device
    command = b'CMD_STATUS\n'  # Must be bytes
    print(f"\n📤 Sending: {command.decode().strip()}")
    ep_out.write(command)

    # Step 8: Read the response
    print("📥 Reading response...")
    response = ep_in.read(64, timeout=1000)  # Read up to 64 bytes
    decoded_response = response.tobytes().decode('utf-8').strip()
    print(f"✅ Response: {decoded_response}")

    # Step 9: Release the device (important!)
    usb.util.release_interface(device, interface)
    print("\n✅ Interface released and USB communication complete.")

except ValueError as e:
    print(f"🔍 Device Not Found: {e}")
except usb.core.USBError as e:
    print(f"🚫 USB Error: {e}")
    print("💡 Tip: Try running with sudo or install correct drivers (e.g.,
Zadig on Windows).")
except Exception as e:
    print(f"🔥 Unexpected Error: {e}")

print("\nℹ️ NOTE: This example requires a real USB device and may need admin
permissions.")
```

7. Key Takeaways

This session provided a hands-on introduction to serial (UART) communication using pyserial, and a conceptual overview of direct USB communication with PyUSB, vital skills for interacting with embedded systems and hardware.

- Serial Communication Basics: Understood the fundamentals of serial data transfer (bit-by-bit), UART, and key parameters like baud rate, data bits, parity, and stop bits.
- COM Ports: Learned about how operating systems identify serial ports (e.g., COM ports on Windows, tty devices on Linux/macOS).
- pyserial Setup: Mastered installing pyserial, listing available ports, and opening/configuring a serial connection using serial.Serial().
- Safe Port Handling: Emphasized the importance of using the with statement for pyserial connections to ensure ports are properly closed.
- Reading & Writing Data (UART): Gained practical experience in sending data (ser.write() - always bytes!) and receiving data (ser.read(), ser.readline()) from a serial port, including converting between strings and bytes.
- Continuous Logging (UART): Conceptualized a real-world scenario of continuously reading sensor data from a device and logging it to a file.

- Microcontroller Integration (Conceptual): Understood how Python with pyserial serves as the bridge for communicating with microcontrollers for control and data acquisition.
- Direct USB Communication (PyUSB):
  - Understood that PyUSB is for devices that don't appear as serial ports, requiring direct access to USB endpoints.
  - Learned about its prerequisites (libusb) and the complexities of finding devices (usb.core.find()) and interacting with their specific endpoints.
  - Gained a conceptual understanding of the low-level workflow for reading from and writing to USB devices.

By mastering pyserial and understanding the principles of PyUSB, you gain the ability to directly interface with a wide range of hardware, enabling you to build powerful automation, data acquisition, and control systems for embedded and IoT applications in engineering and scientific fields.
