Project Report: Retrieving USB & Sensor Data

Aayush Rajput Amitesh Patra

September 2, 2024

Contents

1 Introduction

1.1 Prerequisites

To successfully automate the process of retrieving data from a machine and saving it as a .csv or .json file, several prerequisites need to be met. These prerequisites ensure that you have the necessary tools, knowledge, and environment to carry out the project effectively. Here's a list of key prerequisites:

1. **Understanding of the Machine**
   - **Machine Documentation:** Obtain and thoroughly review the machine's user manual, technical specifications, and communication interface documentation. This helps you understand how to interact with the machine and what data can be retrieved.
   - **Access to Machine Interface:** Ensure you have physical access to the machine and can connect it to your computer or network for testing and data retrieval.
2. **Knowledge of Communication Protocols**
   - **Protocol Basics:** Familiarize yourself with the communication protocols supported by the machine (e.g., Serial, USB, Ethernet, HTTP, Modbus, OPC UA).
   - **Protocol-Specific Tools:** Acquire any necessary tools or software libraries for implementing these protocols (e.g., `pySerial` for serial communication in Python).
3. **Programming Skills**
   - **Language Proficiency:** Be proficient in a programming language suitable for the project (e.g., Python, Java, C++). Python is often preferred due to its extensive library support and ease of use.
   - **Library Knowledge:** Know how to use libraries for data communication and file handling in your chosen programming language (e.g., `usb.core`, `requests`, `csv`, `json` in Python).
4. **Hardware and Software Setup**
   - **Development Environment:** Set up a development environment on your computer that includes the necessary programming tools, compilers, or interpreters.
   - **Drivers and Interfaces:** Install the appropriate drivers and interfaces for connecting the machine to your computer (e.g., USB drivers, Ethernet adapters).
   - **Data Acquisition Hardware (if needed):** Ensure you have any additional hardware required for data acquisition, such as USB-to-serial converters or Ethernet-to-USB adapters.
5. **Data Storage and Analysis Tools**
   - **Data Storage Solutions:** Determine where and how you will store the retrieved data (e.g., local file system, cloud storage, databases).
   - **Analysis Tools:** Identify and install any tools you may need for analyzing the data after it has been logged (e.g., Excel, MATLAB, Python data analysis libraries like `pandas`).
6. **Basic Networking Knowledge**
   - **Networking Basics:** If using network-based protocols (like Ethernet), understand basic networking concepts, including IP addressing, subnetting, and how to configure network devices.
   - **Network Setup:** Ensure that your network is properly configured to allow communication between the machine and your data logging system.
7. **Error Handling and Debugging Skills**
   - **Debugging Tools:** Familiarize yourself with debugging tools and techniques to troubleshoot issues that arise during communication with the machine or during data processing.
   - **Error Handling:** Be prepared to implement robust error handling in your software to manage communication failures, data corruption, or unexpected machine responses.
8. **Time Synchronization**
   - **System Time Configuration:** Ensure that the system time on your data logging device is accurate and synchronized, especially if precise timestamps are critical for your application.
9. **Testing Environment**
   - **Test Setup:** Set up a test environment where you can safely experiment with data retrieval and automation without affecting the actual machine's operation.

- **Sample Data:** If possible, obtain sample data from the machine to use in testing your software and data processing routines.

10. **Project Management Skills**
    - **Task Planning:** Plan the project tasks, including milestone setting and resource allocation, to ensure timely completion.
    - **Version Control:** Use version control systems (e.g., Git) to manage changes to your codebase and collaborate with others if necessary.

11. **Security Considerations**
    - **Data Security:** Ensure that any data transmission is secure, especially if dealing with sensitive or proprietary information. Consider encryption methods if necessary.
    - **Access Control:** Implement access control measures to prevent unauthorized access to the machine or data logging system.

2 Literature Review

2.1 Overview

Automating the process of retrieving data from a machine and saving it as a .csv or .json file is a critical task that can be applied across various industries and research fields. The exact implementation can vary depending on the type of machine, the communication protocols it supports, and the operational environment. Below, we'll explore the general steps involved in this process and then discuss specific examples, such as streaming data from a weighing machine to a computer via wired means, and the role of data logging in scientific machines.

2.2 Steps to Automate Data Retrieval and Storage

2.2.1 Understand the Machine

Begin by thoroughly understanding the machine from which you want to retrieve data. This includes identifying the data sources within the machine, the types of data it generates, and any built-in capabilities it has for communication. Understanding the machine's specifications and interfaces will guide the choice of communication protocols and data retrieval methods.

2.2.2 Select Communication Protocol

Depending on the machine, choose an appropriate communication protocol for data retrieval. Common protocols include:

- **HTTP/HTTPS:** Often used for web-based interfaces or APIs.
- **MQTT (Message Queuing Telemetry Transport):** Useful for low-bandwidth, high-latency environments, and widely used in IoT applications.
- **Modbus:** A serial communication protocol often used in industrial settings.
- **OPC UA (Open Platform Communications Unified Architecture):** Used for secure and reliable data exchange in industrial automation.
- **Serial Communication (RS-232/RS-485/USB):** Common for direct, low-level machine interfaces.
- **Ethernet:** For networked devices that require fast, high-volume data transmission.

2.2.3 Develop Data Retrieval Software

Write software or scripts that can communicate with the machine using the selected protocol. This might involve using programming languages like Python, Java, or C++. The software should be able to establish a connection with the machine, request or receive data, and handle the data appropriately.

Example: In Python, libraries like `requests` can be used for HTTP communication, while `pySerial` can be used for serial communication.

2.2.4 Format Data

Once data is retrieved, format it into .csv or .json as required. This involves structuring the data in a tabular format (for .csv) or a nested key-value format (for .json). Proper formatting ensures that the data is easy to analyze and can be imported into other tools or databases.

2.2.5 Automate Data Saving

Implement automation to periodically retrieve data from the machine and save it to a .csv or .json file. This can be achieved by scheduling tasks using tools like cron jobs (on Unix-like systems) or Task Scheduler (on Windows). Alternatively, you could implement a continuous monitoring system within your software to constantly retrieve and save data.

2.2.6 Handle Errors and Exceptions

Develop robust error handling to manage potential issues during data retrieval and saving. This includes dealing with network errors, timeouts, unexpected data formats, or machine unavailability. Proper logging and alerting mechanisms should be included to notify you of issues as they occur.

2.2.7 Testing and Validation

Thoroughly test your automation solution in various scenarios to ensure it works reliably under different conditions. Validate the output data to ensure accuracy and completeness. This might involve comparing retrieved data with known values or conducting performance tests under load.

2.2.8 Deployment and Monitoring

Once testing is complete, deploy your automation solution in the production environment. Regular monitoring is essential to ensure that the system continues to function correctly over time. Set up alerts for any anomalies or failures, and periodically review the system's performance for potential optimizations.

3 Streaming Data from a Weighing Machine to a Computer via Wired Means

Streaming data from a weighing machine to a computer involves real-time data transmission using a wired connection. The specific protocol and setup will depend on the weighing machine's interface and the requirements of the data logging system.

3.1 Key Steps and Considerations

Weighing Machine Interface: Ensure that the weighing machine has an appropriate interface for wired communication. Common interfaces include:

- **Serial Ports (RS-232/RS-485):** Often used in industrial equipment for direct communication.
- **USB:** Common for more modern machines, offering plug-and-play capabilities.
- **Ethernet:** For networked machines, allowing them to communicate over local or wide-area networks.

Data Acquisition System or Data Logger: Use a data acquisition system or a data logger that supports the chosen communication protocol. These devices or systems are responsible for receiving data from the weighing machine and transmitting it to the computer.
Examples:

- **National Instruments:** Known for high-quality data acquisition systems.
- **DataLogix or Campbell Scientific:** Offer robust data logging solutions for various industrial applications.

Computer Interface: Ensure that the computer has the necessary hardware and software to interface with the weighing machine. This might include:

- Serial-to-USB converters for older machines.
- Network interfaces for Ethernet-connected devices.

Data Transmission and Reception: Establish a real-time data transmission system. This involves configuring the weighing machine to continuously send data, setting up the data logger or acquisition system to capture this data, and ensuring that the computer can properly receive and process it.
Protocols: For real-time streaming, consider using low-latency, high-throughput protocols suitable for the data type and volume.

3.1.1 Data Storage and Analysis

Store the incoming data in an organized manner, typically in a database or as sequential .csv or .json files. Ensure that the data structure is optimized for later analysis. Use data analysis tools or software to interpret and visualize the data, enabling insights and decisions based on the machine's output.

3.2 Data Logging in Scientific Machines

Data logging is essential in scientific research, industrial processes, and laboratory environments. It involves continuously recording data from various instruments and machines to track changes over time, identify patterns, and ensure quality control.

3.2.1 Examples of Scientific Machines Using Data Logging

Weighing Machines: Precision balances and other weighing devices log weight data continuously. This is critical in experiments where material weight changes over time, such as in chemical reactions or in quality control processes in manufacturing.
Environmental Monitoring Systems: Systems like weather stations or air quality monitors record environmental parameters (e.g., temperature, humidity, air pressure) over time. This data is essential for tracking climate changes, studying environmental impacts, and ensuring compliance with environmental regulations.
Temperature Data Loggers: Used in applications ranging from food storage to pharmaceutical manufacturing, these devices monitor and record temperature in real-time, ensuring that products are kept within safe temperature ranges.
pH Data Loggers: Essential in water quality monitoring, soil testing, and chemical processes, these loggers record pH levels over time to ensure that environmental or process conditions remain within acceptable parameters.
Strain and Vibration Data Loggers: These are used in mechanical testing and materials science to monitor the stress and vibrations that materials or structures undergo over time. This data helps in understanding material properties and ensuring structural integrity.

Acoustic Data Loggers: These devices record sound levels for applications such as noise monitoring, acoustic research, and compliance with noise regulations.

Light Data Loggers: Used in photometry and optical research, these loggers record light intensity over time, essential for studies that require precise control of light exposure.

Pressure and Flow Data Loggers: These loggers monitor and record pressure and fluid flow in various industrial and research applications. Accurate data logging is crucial for fluid dynamics studies and process control.

3.3 Conclusion

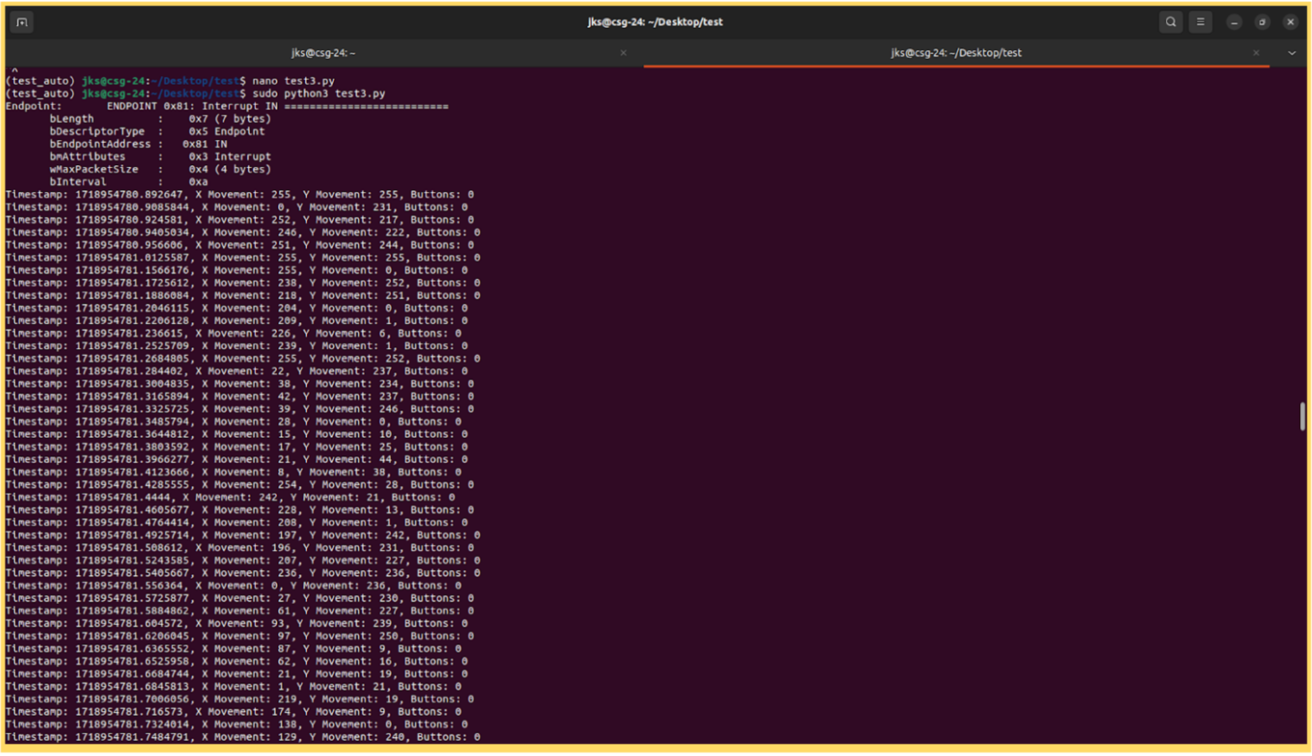
Automating data retrieval and logging from machines to .csv or .json files is a multifaceted process that involves understanding the machine's communication capabilities, selecting the right protocols, developing appropriate software, and ensuring reliable operation. By following a structured approach and considering the specific requirements of the machine and the data logging environment, you can create a robust and efficient data logging system.

4 Real-Time USB Mouse Data Capture and Logging Using Python and pyUSB

4.1 Overview

This Python script is designed to capture and record data from a USB mouse in real-time using the `pyusb` library. It starts by identifying the USB device based on its vendor and product IDs. Once the device is found, it detaches any kernel drivers that might be managing the device to ensure direct control. The script then sets the device configuration and retrieves the first data endpoint from the device's first interface, which is typically used for transferring data like mouse movements and button presses.
To record the mouse data, the script creates a CSV file named `mouse_data.csv` and writes headers for timestamp, X movement, Y movement, and button states. It then enters an infinite loop where it continuously reads data from the mouse's endpoint. This data is parsed to extract button states and movements along the X and Y axes, which are then written to the CSV file along with a timestamp.

The script includes error handling to manage USB read timeouts, ensuring smooth operation. It also prints the captured data to the console for debugging purposes. The loop is designed to run indefinitely, capturing all mouse movements and button presses until manually interrupted. When the script is stopped, the CSV file is closed properly to ensure all data is saved. This approach provides a simple and effective way to log detailed mouse activity.



```
(test_auto) jks@csg-24:~/Desktop/test$ nano test3.py
(test_auto) jks@csg-24:~/Desktop/test$ sudo python3 test3.py
Endpoint:       ENDPOINT 0x81: Interrupt IN ==========================
       bLength          :    0x7 (7 bytes)
       bDescriptorType  :    0x5 Endpoint
       bEndpointAddress :   0x81 IN
       bmAttributes     :    0x3 Interrupt
       wMaxPacketSize   :    0x4 (4 bytes)
       bInterval        :    0xa
Timestamp: 1718954780.892647, X Movement: 255, Y Movement: 255, Buttons: 0
Timestamp: 1718954780.9085844, X Movement: 0, Y Movement: 231, Buttons: 0
Timestamp: 1718954780.924581, X Movement: 252, Y Movement: 217, Buttons: 0
Timestamp: 1718954780.9405034, X Movement: 246, Y Movement: 222, Buttons: 0
Timestamp: 1718954780.956606, X Movement: 251, Y Movement: 244, Buttons: 0
Timestamp: 1718954781.0125587, X Movement: 255, Y Movement: 255, Buttons: 0
Timestamp: 1718954781.1566176, X Movement: 255, Y Movement: 0, Buttons: 0
Timestamp: 1718954781.1725612, X Movement: 238, Y Movement: 252, Buttons: 0
Timestamp: 1718954781.1886084, X Movement: 218, Y Movement: 251, Buttons: 0
Timestamp: 1718954781.2046115, X Movement: 204, Y Movement: 0, Buttons: 0
Timestamp: 1718954781.2206128, X Movement: 209, Y Movement: 1, Buttons: 0
Timestamp: 1718954781.236615, X Movement: 226, Y Movement: 6, Buttons: 0
Timestamp: 1718954781.2525709, X Movement: 239, Y Movement: 1, Buttons: 0
Timestamp: 1718954781.2684805, X Movement: 255, Y Movement: 252, Buttons: 0
Timestamp: 1718954781.284402, X Movement: 22, Y Movement: 237, Buttons: 0
Timestamp: 1718954781.3004835, X Movement: 38, Y Movement: 234, Buttons: 0
Timestamp: 1718954781.3165894, X Movement: 42, Y Movement: 237, Buttons: 0
Timestamp: 1718954781.3325725, X Movement: 39, Y Movement: 246, Buttons: 0
Timestamp: 1718954781.3485794, X Movement: 28, Y Movement: 0, Buttons: 0
Timestamp: 1718954781.3644812, X Movement: 15, Y Movement: 10, Buttons: 0
Timestamp: 1718954781.3803592, X Movement: 17, Y Movement: 25, Buttons: 0
Timestamp: 1718954781.3966277, X Movement: 21, Y Movement: 44, Buttons: 0
Timestamp: 1718954781.4123666, X Movement: 8, Y Movement: 38, Buttons: 0
Timestamp: 1718954781.4285555, X Movement: 254, Y Movement: 28, Buttons: 0
Timestamp: 1718954781.4444, X Movement: 242, Y Movement: 21, Buttons: 0
Timestamp: 1718954781.4605677, X Movement: 228, Y Movement: 13, Buttons: 0
Timestamp: 1718954781.4764414, X Movement: 208, Y Movement: 1, Buttons: 0
Timestamp: 1718954781.4925714, X Movement: 197, Y Movement: 242, Buttons: 0
Timestamp: 1718954781.508612, X Movement: 196, Y Movement: 231, Buttons: 0
Timestamp: 1718954781.5243585, X Movement: 207, Y Movement: 227, Buttons: 0
Timestamp: 1718954781.5405667, X Movement: 236, Y Movement: 236, Buttons: 0
Timestamp: 1718954781.556364, X Movement: 0, Y Movement: 236, Buttons: 0
Timestamp: 1718954781.5725877, X Movement: 27, Y Movement: 230, Buttons: 0
Timestamp: 1718954781.5884862, X Movement: 61, Y Movement: 227, Buttons: 0
Timestamp: 1718954781.604572, X Movement: 93, Y Movement: 239, Buttons: 0
Timestamp: 1718954781.6206045, X Movement: 97, Y Movement: 250, Buttons: 0
Timestamp: 1718954781.6365552, X Movement: 87, Y Movement: 9, Buttons: 0
Timestamp: 1718954781.6525958, X Movement: 62, Y Movement: 16, Buttons: 0
Timestamp: 1718954781.6684744, X Movement: 21, Y Movement: 19, Buttons: 0
Timestamp: 1718954781.6845813, X Movement: 1, Y Movement: 21, Buttons: 0
Timestamp: 1718954781.7006056, X Movement: 219, Y Movement: 19, Buttons: 0
Timestamp: 1718954781.716573, X Movement: 174, Y Movement: 9, Buttons: 0
Timestamp: 1718954781.7324014, X Movement: 138, Y Movement: 0, Buttons: 0
Timestamp: 1718954781.7484791, X Movement: 129, Y Movement: 240, Buttons: 0
```

Figure 1: Mouse Output

Listing 1: Python Code for USB Mouse Interaction

```
import usb.core
import usb.util
import csv
import time

# Find the USB mouse by HID device class
device = usb.core.find(idVendor=0x0461, idProduct=0x4e2a)

ep = device[0].interfaces()[0].endpoints()[0]

print("ep :", ep)

i = device[0].interfaces()[0].bInterfaceNumber

if device.is_kernel_driver_active(i):
    device.detach_kernel_driver(i)

device.reset()

device.set_configuration()
eaddr = ep.bEndpointAddress

r = device.read(eaddr, 1024)
print(r)
```

The script includes error handling to manage USB read timeouts, ensuring smooth operation. It also prints the captured data to the console for debugging purposes. The loop is designed to run indefinitely, capturing all mouse movements and button presses until manually interrupted. When the script is stopped, the CSV file is closed properly to ensure all data is saved. This approach provides a simple and effective way to log detailed mouse activity.

Listing 2: Real-Time USB Mouse Data Capture and Logging

```python
import usb.core
import usb.util
import time
import csv

# Vendor and Product IDs for the USB mouse (replace with your specific mouse IDs)
vendor_id = 0x0461
product_id = 0x4e2a

# Find the USB device by Vendor ID and Product ID
device = usb.core.find(idVendor=vendor_id, idProduct=product_id)

if device is None:
    raise ValueError('Device not found')

# Detach kernel driver if active
for cfg in device:
    for intf in cfg:
        if device.is_kernel_driver_active(intf.bInterfaceNumber):
            device.detach_kernel_driver(intf.bInterfaceNumber)

# Set the active configuration. This is usually not necessary for HID devices but ensures c
device.set_configuration()

# Get the first endpoint of the first interface (assuming it's the one for data)
endpoint = device[0][(0, 0)][0]

# Print endpoint information for debugging
print("Endpoint:", endpoint)

# Create a CSV file to save the mouse data
csv_file = open('mouse_data.csv', mode='w', newline='')
csv_writer = csv.writer(csv_file)

# Write headers to CSV file (optional)
csv_writer.writerow(['Timestamp', 'X-Movement', 'Y-Movement', 'Buttons'])

# Continuously read data from the endpoint and write to CSV
try:
    while True:
        try:
            # Attempt to read data from the endpoint
            data = device.read(endpoint.bEndpointAddress,
            endpoint.wMaxPacketSize, timeout=100)

            # Parse and interpret the data (assuming it's mouse input data)
            button_state = data[0]
            x_movement = data[1]
            y_movement = data[2]

            # Get current timestamp
            timestamp = time.time()

            # Write data to CSV file
            csv_writer.writerow([timestamp, x_movement, y_movement,
                                  button_state])

            # Print for debugging (optional)
            print(f"Timestamp: {timestamp}, X Movement: {x_movement},
            Y Movement: {y_movement}, Buttons: {button_state}")

        except usb.core.USBError as e:
            if e.args == ('Operation timed out',):
                continue

        # Sleep briefly to avoid high CPU usage in the loop
        time.sleep(0.01)

except KeyboardInterrupt:
    print("Execution interrupted by user.")

finally:
    # Close the CSV file
    csv_file.close()
```

5 Code Snippets

Listing 3: Explanation of the Diagram Steps

```
Start: The script begins with initialization.
Find USB Device: The script attempts to find the USB device using the provided Vendor ID anc
                 If the device is not found, an error is raised.
Detach Kernel Driver: If the device is found, it detaches the kernel driver to allow direct
Set Device Configuration: The script sets the active configuration to ensure proper operatio
Get First Endpoint: The first endpoint of the device is retrieved, and its information is p
Start Data Capture Loop: The script enters an infinite loop to continuously capture data fro
Read Data from Endpoint: Data is read from the endpoint. If successful, the data is parsed a
                         If a timeout error occurs, the script handles it and continues the l
```

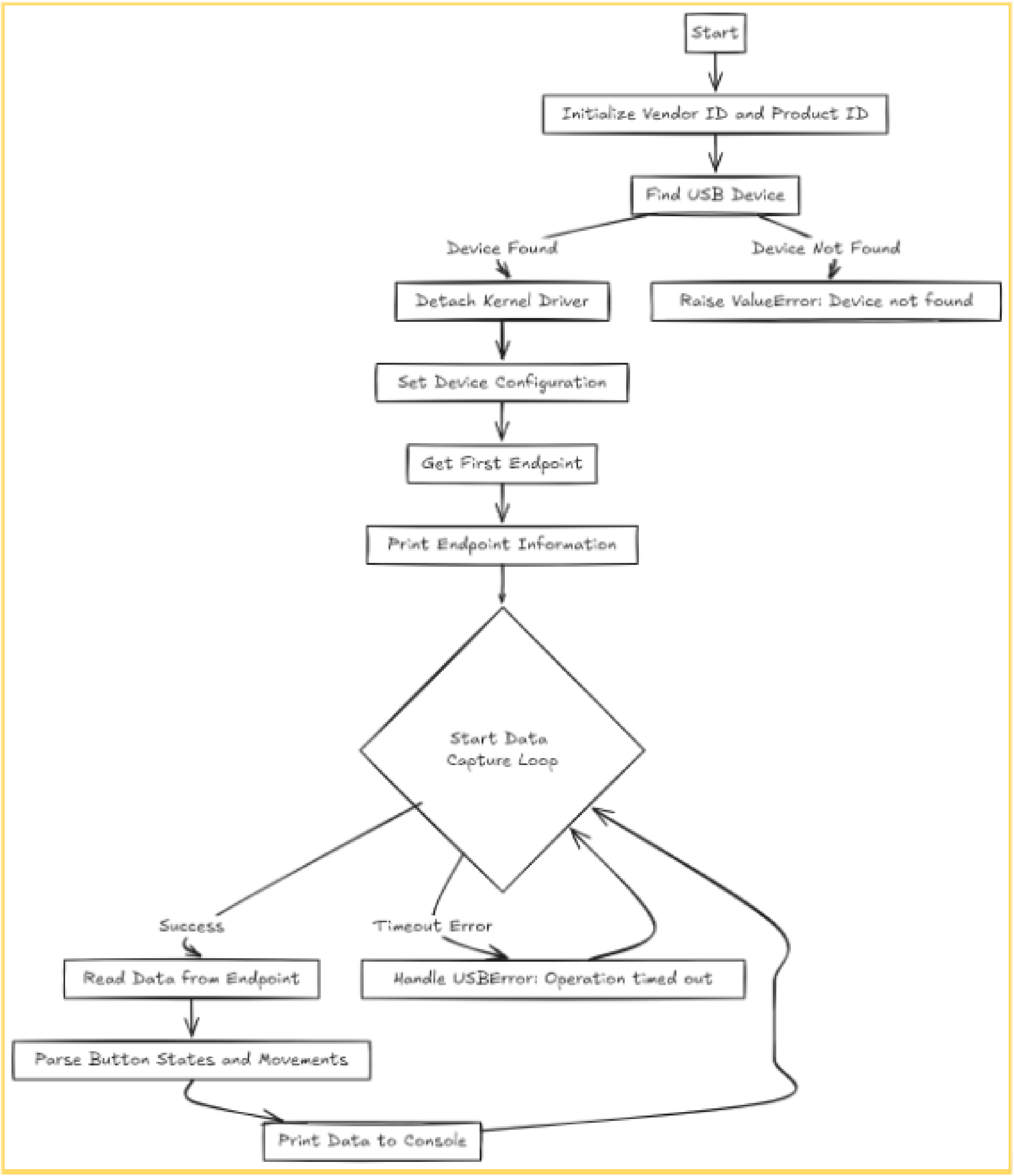


Figure 2: General Architecture of Retrieving Mouse data

Listing 4: Real-Time USB Mouse Data Capture

```
import usb.core
import usb.util
import time

# Vendor and Product IDs for the USB mouse (replace with your specific mouse IDs)
vendor_id = 0x0461
product_id = 0x4e2a

# Find the USB device by Vendor ID and Product ID
device = usb.core.find(idVendor=vendor_id, idProduct=product_id)

if device is None:
    raise ValueError('Device not found')

# Detach kernel driver if active
for cfg in device:
    for intf in cfg:
        if device.is_kernel_driver_active(intf.bInterfaceNumber):
            device.detach_kernel_driver(intf.bInterfaceNumber)

# Set the active configuration. This is usually not necessary for HID devices but ensures c
device.set_configuration()

# Get the first endpoint of the first interface (assuming it's the one for data)
endpoint = device[0][(0, 0)][0]

# Print endpoint information for debugging
print("Endpoint:", endpoint)

# Continuously read data from the endpoint
while True:
    try:
        # Attempt to read data from the endpoint
        data = device.read(endpoint.bEndpointAddress, endpoint.wMaxPacketSize, timeout=100)

        # Print raw data for debugging
        print("Raw data:", data)

        # Parse and interpret the data (assuming it's mouse input data)
        # Example interpretation:
        button_state = data[0]  # First byte often represents button states
        x_movement = data[1]    # Second byte can represent movement along X axis
        y_movement = data[2]    # Third byte can represent movement along Y axis

        # Example output (print movement and button state)
        print(f"X Movement: {x_movement}, Y Movement: {y_movement}, Buttons: {button_state}"

        # Optional: Process the data further (e.g., log to file, analyze gestures)

    except usb.core.USBError as e:
        if e.args == ('Operation timed out',):
            continue

    # Sleep briefly to avoid high CPU usage in the loop
    time.sleep(0.01)
```

This diagram visually represents the sequence and logic flow of the code in a simplified manner.

Listing 5: Real-Time USB Mouse Data Capture

```
import usb.core
import usb.util
import time

# Vendor and Product IDs for the USB mouse (replace with your specific mouse IDs)
vendor_id = 0x0461
product_id = 0x4e2a

# Find the USB device by Vendor ID and Product ID
device = usb.core.find(idVendor=vendor_id, idProduct=product_id)

if device is None:
    raise ValueError('Device not found')

# Detach kernel driver if active
for cfg in device:
    for intf in cfg:
        if device.is_kernel_driver_active(intf.bInterfaceNumber):
            device.detach_kernel_driver(intf.bInterfaceNumber)

# Set the active configuration.
This is usually not necessary for HID devices but ensures correct operation.
device.set_configuration()

# Get the first endpoint of the
# first interface (assuming it's the one for data)
endpoint = device[0][(0, 0)][0]

# Print endpoint information for debugging
print("Endpoint:", endpoint)

# Continuously read data from the endpoint
while True:
    try:
        # Attempt to read data from the endpoint
        data = device.read(endpoint.bEndpointAddress, endpoint.wMaxPacketSize, timeout=100)

        # Print raw data for debugging
        print("Raw data:", data)

        # Parse and interpret the data (assuming it's mouse input data)
        # Example interpretation:
        button_state = data[0]  # First byte often represents button states
        x_movement = data[1]    # Second byte can represent movement along X axis
        y_movement = data[2]    # Third byte can represent movement along Y axis

        # Example output (print movement and button state)
        print(f"X Movement: {x_movement}, Y Movement: {y_movement}, Buttons: {button_state}"

        # Optional: Process the data further (e.g., log to file, analyze gestures)

    except usb.core.USBError as e:
        if e.args == ('Operation timed out',):
            continue

    # Sleep briefly to avoid high CPU usage in the loop
    time.sleep(0.01)
```

6 Real-Time Gyroscope Data Collection and Analysis System

6.1 Overview

This chapter presents a system designed to capture and analyze gyroscope data from an Android device, transmitting it to a Python server for storage and subsequent analysis. The project exemplifies the integration of mobile and server-side technologies, with a focus on real-time data handling. The system is composed of two primary components:
Android Application: Responsible for capturing gyroscope sensor data and transmitting it to the server.
Python Server: Receives the data and stores it in a CSV file for further analysis.

6.2 Android Application

Structure and Functionality:
The Android application, written in Kotlin, employs modern Android development practices, including Jetpack Compose for the user interface. The key functionalities of the application include:

6.2.1 Sensor Management:

Utilizes Android's SensorManager to access the gyroscope sensor. Implements the SensorEventListener interface to handle real-time sensor data, triggering the onSensorChanged method with each update.

- sensorManager = getSystemService(SENSOR SERVICE) as SensorManager
- gyroscopeSensor = sensorManager.getDefaultSensor(Sensor.TYPE GYROSCOPE)

6.2.2 Handling Sensor Data:

Captures gyroscope data along the x, y, and z axes, which is then processed and prepared for transmission.

```
override fun onSensorChanged(event: SensorEvent) {
    if (event.sensor.type == Sensor.TYPE_GYROSCOPE) {
        val x = event.values[0]
        val y = event.values[1]
        val z = event.values[2]
        gyroscopeData = "Gyroscope Data:\nX: $x\nY: $y\nZ: $z"
        coroutineScope.launch {
            sendGyroscopeData(x, y, z)
        }
    }
}
```

6.2.3 Networking:

- Uses Retrofit, a type-safe HTTP client, to send gyroscope data to the Python server.
- OkHttpClient is integrated for monitoring network requests, assisting in debugging.

```
private suspend fun sendGyroscopeData(x: Float, y: Float, z: Float) {
    val data = GyroscopeData(x, y, z)
    try {
        apiService.sendGyroscopeData(data).execute()
    } catch (e: Exception) {
        e.printStackTrace()
    }
}
```

6.2.4 User Interface:

The UI, built with Jetpack Compose, displays gyroscope data in real-time.

```
@Composable
fun GyroscopeDataDisplay(gyroscopeData: String) {
    Text(text = gyroscopeData)
}
```

7 Python Server

Structure and Functionality:

The Python server is a Flask-based application that handles incoming POST requests containing gyroscope data from the Android application. Key functions include:

7.1 Data Handling:

- Listens for POST requests at the /gyroscope endpoint.
- Extracts x, y, and z axis values from the received JSON payload.

7.2 CSV Storage:

- Appends the data to gyroscope data.csv, ensuring that headers (X, Y, Z) are included if the file is newly created.

```
if not os.path.exists(csv_file):
    with open(csv_file, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['X', 'Y', 'Z'])
```

7.3 Receiving and Storing Data:

```
@app.route('/gyroscope', methods=['POST'])
def gyroscope():
    data = request.get_json()
    x = data.get('x')
    y = data.get('y')
    z = data.get('z')

    with open(csv_file, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([x, y, z])

    return jsonify({"message": "Data received"}), 200
```

7.4 Running the Server:

```
if __name__ == '__main__':
    app.run(host='127.0.0.1', port=5001)
```

8 Integration and Workflow:

8.1 Initialization:

- The Android app initializes the gyroscope sensor and Retrofit client.
- The Flask server initializes the CSV file and begins listening for incoming data.

8.2 Data Collection:

- The gyroscope sensor captures rotational motion data as the user moves the Android device, with the data displayed on the device screen in real-time.

8.3 Data Transmission:

- The Android application sends the gyroscope data to the Python server via Retrofit, packaged as a JSON object in a POST request.

8.4 Data Storage:

- The Python server extracts the data and appends it to the gyroscope data.csv file for further analysis or visualization.

9 Android Application Implementation:

The Android application, written in Kotlin, serves as the client-side component of the gyroscope data collection system. It captures real-time data from the device's gyroscope sensor and sends this data to a Python server. Below is the detailed breakdown of the code implementation:

Listing 6: Implementation of the Gyroscope Data Collection System

```
package com.example.gyroscopereaderwithpythonserver

import android.hardware.Sensor
import android.hardware.SensorEvent
import android.hardware.SensorEventListener
import android.hardware.SensorManager
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import com.example.gyroscopereaderwithpythonserver.ui.theme.GyroscopeReaderTheme
import kotlinx.coroutines.*
import okhttp3.OkHttpClient
import okhttp3.logging.HttpLoggingInterceptor
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

class MainActivity : ComponentActivity(), SensorEventListener {
    private lateinit var sensorManager: SensorManager
    private var gyroscopeSensor: Sensor? = null
    private var gyroscopeData by mutableStateOf("Gyroscope Data:\nX: 0.0\nY: 0.0\nZ: 0.0")
    private lateinit var apiService: ApiService
    private val coroutineScope = CoroutineScope(Dispatchers.IO)

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Initialize sensor manager and gyroscope sensor
        sensorManager = getSystemService(SENSOR_SERVICE) as SensorManager
        gyroscopeSensor = sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE)

        // Setup HTTP client and Retrofit for network communication
        val logging = HttpLoggingInterceptor().apply {
            level = HttpLoggingInterceptor.Level.BODY
        }
        val client = OkHttpClient.Builder()
            .addInterceptor(logging)
            .build()

        val retrofit = Retrofit.Builder()
            .baseUrl("http://127.0.0.1:5001/")
            .addConverterFactory(GsonConverterFactory.create())
            .client(client)
            .build()

        apiService = retrofit.create(ApiService::class.java)

        // Set the content view to display gyroscope data using Jetpack Compose
        setContent {
            GyroscopeReaderTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    GyroscopeDataDisplay(gyroscopeData)
                }
            }
        }
    }

    override fun onResume() {
        super.onResume()
        gyroscopeSensor?.also { sensor ->
            sensorManager.registerListener(this, sensor, SensorManager.SENSOR_DELAY_NORMAL)
        }
    }

    override fun onPause() {
        super.onPause()
        sensorManager.unregisterListener(this)
    }

    // Handle gyroscope data when sensor values change
    override fun onSensorChanged(event: SensorEvent) {
        if (event.sensor.type == Sensor.TYPE_GYROSCOPE) {
            val x = event.values[0]
            val y = event.values[1]
            val z = event.values[2]
            gyroscopeData = "Gyroscope Data:\nX: $x\nY: $y\nZ: $z"

            // Send gyroscope data to the server asynchronously
            coroutineScope.launch {
                sendGyroscopeData(x, y, z)
            }
        }
    }

    override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {
        // No implementation required for this project
    }

    private suspend fun sendGyroscopeData(x: Float, y: Float, z: Float) {
        val data = GyroscopeData(x, y, z)
        try {
            apiService.sendGyroscopeData(data).execute()
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}

// Composable function to display gyroscope data in the UI
@Composable
fun GyroscopeDataDisplay(data: String) {
    Column(
        modifier = Modifier.fillMaxSize()
    ) {
        Text(
            text = data,
            style = MaterialTheme.typography.bodyLarge
        )
    }
}

// Preview function for the GyroscopeDataDisplay composable
@Preview(showBackground = true)
```

```
@Composable
fun DefaultPreview() {
    GyroscopeReaderTheme {
        GyroscopeDataDisplay("Gyroscope Data:\nX: 0.0\nY: 0.0\nZ: 0.0")
    }
}
```

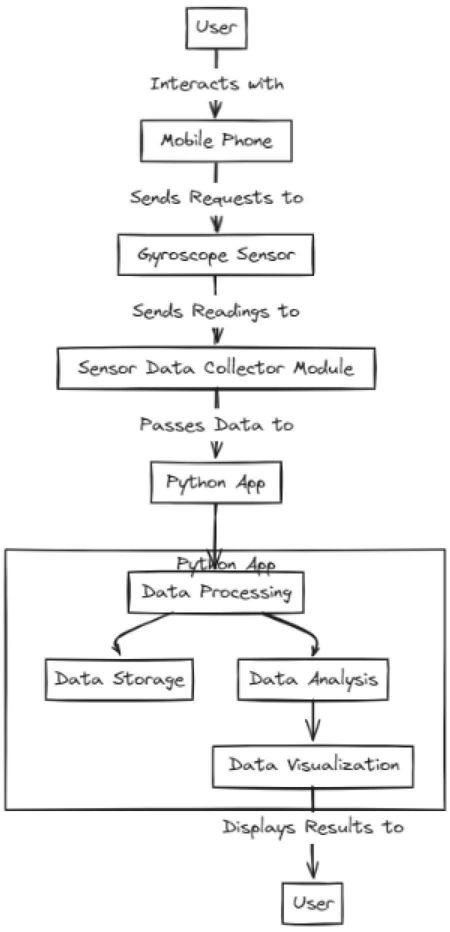


Figure 3: Architecture

9.1 Diagram Explaination:

This diagram illustrates the flow of data from a user interacting with a mobile phone's gyroscope sensor to the final presentation of processed results. Here's a detailed explanation of each step:

- User: The process begins with the user, who interacts with their mobile phone, possibly through an application designed to collect gyroscope data.
- Mobile Phone: The mobile phone acts as the hardware platform where the user's interactions are translated into sensor data requests. This could be through actions like tilting, rotating, or moving the phone.
- Gyroscope Sensor: The gyroscope sensor in the mobile phone captures the orientation and rotational movement data. This sensor detects the rate of rotation around the three physical axes (x, y, and z).
- Sensor Data Collector Module: This module, likely part of the Android application, collects the raw data from the gyroscope sensor. It processes the readings to ensure they are in a suitable format for transmission and further processing.
- Python App: The collected sensor data is then passed from the mobile application to a Python application running on a server. This transfer happens over a network, typically using HTTP requests.
- Data Processing: Once the Python app receives the data, it enters the data processing phase. Here, the data is prepared for storage, analysis, or both. Processing may involve cleaning, filtering, or converting the data into a specific format.

- Data Storage: After processing, the data is stored, typically in a CSV file or a database. This stored data can be used for future analysis, allowing historical data trends to be observed.
- Data Analysis: The Python app also performs analysis on the gyroscope data. This might involve computing statistics, detecting patterns, or identifying specific movements. The results of this analysis provide insights into the user's movements.
- Data Visualization: The analyzed data is then visualized, possibly through graphs, charts, or other visual formats. This visualization helps in interpreting the data easily and is presented back to the user.
- Displays Results to User: Finally, the processed and visualized results are displayed back to the user, closing the loop. The user can then view the results of their movements as captured and processed by the system.

10 Conclusion:

This chapter illustrates the successful integration of Android and Python technologies to create a robust system for real-time gyroscope data collection and storage. The system is flexible and can be extended to support various applications, such as motion tracking, gesture recognition, or gaming, providing a foundation for further innovation in mobile and server-side development.