

Project Report

DIXELL -> JETSON NANO -> AWS CLOUD

01-02-2024

Contents

Overview:.....	3
Camera Integration and Video Streaming:	3
G-Streamer Plugin Stream Configurations:	3
Sample GStreamer Pipeline for Kinesis Stream:	3
Integration of KVSSINK CPP SDK for Jetson Nano:	3
KVS Sink Pipelines:	4
System Variable Setup and Testing:	4
Script for Video Streaming to Kinesis:	4
Script Overview:	4
Explanation:	5
Output Visualization:.....	5
Local Jetson Nano Command Window Output:.....	5
Kinesis AWS Video Stream Output:.....	6
ESP32 Wireless Conversion for Modbus Communication:.....	6
Integration Overview:	6
ESP32 Setup:	6
Data Parsing and Packaging:	6
Wireless Data Transmission:	6
Jetson Nano Reception and AWS Integration:	7
ESP32 Wireless Data Reception on Jetson Nano:	7
Script Overview:	7
Integration Steps:	8
AWS Cloud Configurations:	8
MQTT Client Setup:	8
CSV File Handling:	8
Data Processing and Publishing:	9
ESP32 Bluetooth Communication:	9
Wireless Reception and CSV Update:.....	9
Jetson Nano Command Line Output:.....	9
Updated CSV File:.....	9
Final Remarks:.....	10

Project Report

Overview:

This report outlines a project merging USB camera integration with the Jetson Nano, NVIDIA GStreamer SDK utilization for local and cloud video streaming, and a subsequent shift to wireless Modbus communication via ESP32 and Bluetooth. Aimed at providing a technical reference, the document details the project's architecture, implementation steps, and potential troubleshooting points for engineers and collaborators

Camera Integration and Video Streaming:

In this section, we detail the steps taken to integrate a USB camera with the Jetson Nano (/dev/USB0) and implement video streaming using the NVIDIA GStreamer SDK. The progress log outlines the specific configurations and processes undertaken throughout this phase.

G-Streamer Plugin Stream Configurations:

- Utilized GStreamer pipelines to configure and fine-tune the video streaming process for optimal performance.
- Configured pipelines for MJPG to H.264 conversion with varying resolutions and frame rates.
- Implemented a direct MJPG 30fps pipeline for reference.
- Integration of AWS-Kinesis-SDK Docker Container:
- Explored the possibility of Docker support for AWS-Kinesis-SDK on the Jetson Nano, attempting to build the Docker version using ARM-converted libraries.
- Faced challenges with direct Docker support on Jetson Nano due to architecture constraints.
- Resorted to utilizing the Git-pulled SDK, setting it up manually, and incorporating it into the project.

Sample GStreamer Pipeline for Kinesis Stream:

- Configured a GStreamer pipeline for Kinesis streaming with a USB camera.
- Used the v4l2src element to capture video frames.
- Employed tee to split the video stream into two paths.
- Utilized videoconvert, ximagesink, and other elements for processing and visualization.

Integration of KVSSINK CPP SDK for Jetson Nano:

- Configured the Kinesis Producer on the Jetson Nano.
- Set up GStreamer for Kinesis streaming and integrated SDP transmission of USB camera feed to an RTSP server for real-time viewing.
- Explored the KVSSINK C++ SDK for Jetson Nano, inspecting its capabilities and properties.

KVS Sink Pipelines:

- Established KVS sink pipelines for the GStreamer SDK.
- Pipelined the Kinesis SDK to KVS service for secure and efficient video streaming.
- Introduced I420 conversion for enhanced streaming and codec compatibility.

System Variable Setup and Testing:

- Added system variables for secure storage of access keys.
- Tested live streaming on KVS with specific characteristics:
- Video format: I420, resolution: 640x480, framerate: 25fps.
- Monitored Kinesis Video client and stream metrics for performance evaluation.

Script for Video Streaming to Kinesis:

In this section, we present the Python script designed for streaming video from a USB camera on the Jetson Nano to the Kinesis Video Stream on AWS. The script utilizes the Amazon Kinesis Video Streams Producer SDK for C++, AWS IoT Core for MQTT communication, and OpenCV for camera handling.

Script Overview:

```
import cv2

import base64

from AWSIoTPythonSDK.MQTTLib import AWSIoTClient

import ssl

# AWS IoT Core Configurations

# ... (Root CA, certificate, private key, client ID, topic, and endpoint configurations)

def on_connect(client, userdata, flags, rc):

    # ... (Callback function on successful connection)

def on_message(client, userdata, message):

    # ... (Callback function on receiving MQTT message)

def publish_frame_to_iot(frame, client):

    # ... (Encode and publish frames to AWS IoT Core)

def show_camera(client):

    # ... (Configure and display the camera feed, publish frames to AWS IoT Core)

if __name__ == "__main__":

    # MQTT Client Setup

    # ... (Configure AWS IoT Core connection parameters)

    # Connect to AWS IoT Core

    print("Connecting to AWS IoT Core...")

    mqtt_client.connect()
```

```
# Start the camera feed and publish frames to AWS IoT Core

show_camera(mqtt_client)
```

Explanation:

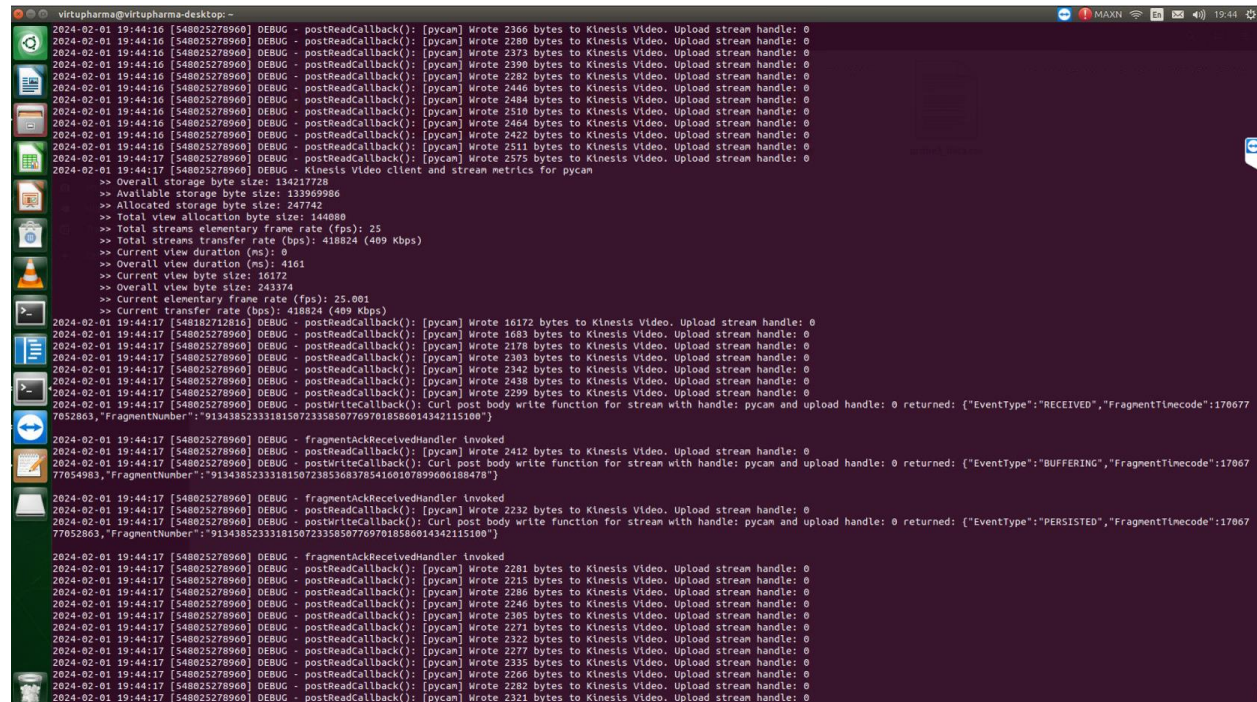
The script establishes a connection to AWS IoT Core using the configured MQTT client.

- It captures video frames from a USB camera using OpenCV, encodes them as JPEG, and publishes them to the specified AWS IoT Core topic.
- The show_camera function continuously displays the camera feed locally and publishes frames to AWS IoT Core, ensuring real-time streaming.

Output Visualization:

The script outputs two visual components: one for the local Jetson Nano command window and another for the Kinesis AWS Video Stream. These outputs provide a comprehensive view of the script's functionality and the video streaming process.

Local Jetson Nano Command Window Output:



```
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2366 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2280 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2373 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2390 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2282 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2446 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2484 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2510 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2464 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2422 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:16 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2511 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2575 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - Kinesis Video client and stream metrics for pycan
>> Overall storage byte size: 13421728
>> Available storage byte size: 133969986
>> Allocated storage byte size: 247742
>> Total view allocation byte size: 144088
>> Total streams elementary frame rate (fps): 25
>> Total streams transfer rate (bps): 418824 (409 kbps)
>> Current view duration (ms): 0
>> Overall view duration (ms): 4161
>> Current view byte size: 16172
>> Overall view byte size: 243374
>> Current elementary frame rate (fps): 25.001
>> Current transfer rate (bps): 418824 (409 kbps)
2024-02-01 19:44:17 [548182712016] DEBUG - postReadCallback(): [pycan] Wrote 16172 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 1683 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2178 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2303 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2342 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2438 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2299 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postWriteCallback(): Curl post body write function for stream with handle: pycan and upload handle: 0 returned: {"EventType":"RECEIVED","FragmentTlncode":170677052863,"FragmentNumber":"","91343852333181507233585077697018586014342115100"}
2024-02-01 19:44:17 [548025278960] DEBUG - fragmentAckReceivedHandler invoked
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2412 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postWriteCallback(): Curl post body write function for stream with handle: pycan and upload handle: 0 returned: {"EventType":"BUFFERING","FragmentTlncode":170677054903,"FragmentNumber":"","913438523331815072335850776970185860188478"}
2024-02-01 19:44:17 [548025278960] DEBUG - fragmentAckReceivedHandler invoked
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2232 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postWriteCallback(): Curl post body write function for stream with handle: pycan and upload handle: 0 returned: {"EventType":"PERSISTED","FragmentTlncode":170677052863,"FragmentNumber":"","91343852333181507233585077697018586014342115100"}
2024-02-01 19:44:17 [548025278960] DEBUG - fragmentAckReceivedHandler invoked
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2281 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2215 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2286 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2246 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2305 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2271 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2322 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2277 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2335 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2266 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2282 bytes to Kinesis Video. Upload stream handle: 0
2024-02-01 19:44:17 [548025278960] DEBUG - postReadCallback(): [pycan] Wrote 2321 bytes to Kinesis Video. Upload stream handle: 0
```

Figure 1 Gstreamer Pipeline Output

Kinesis AWS Video Stream Output:

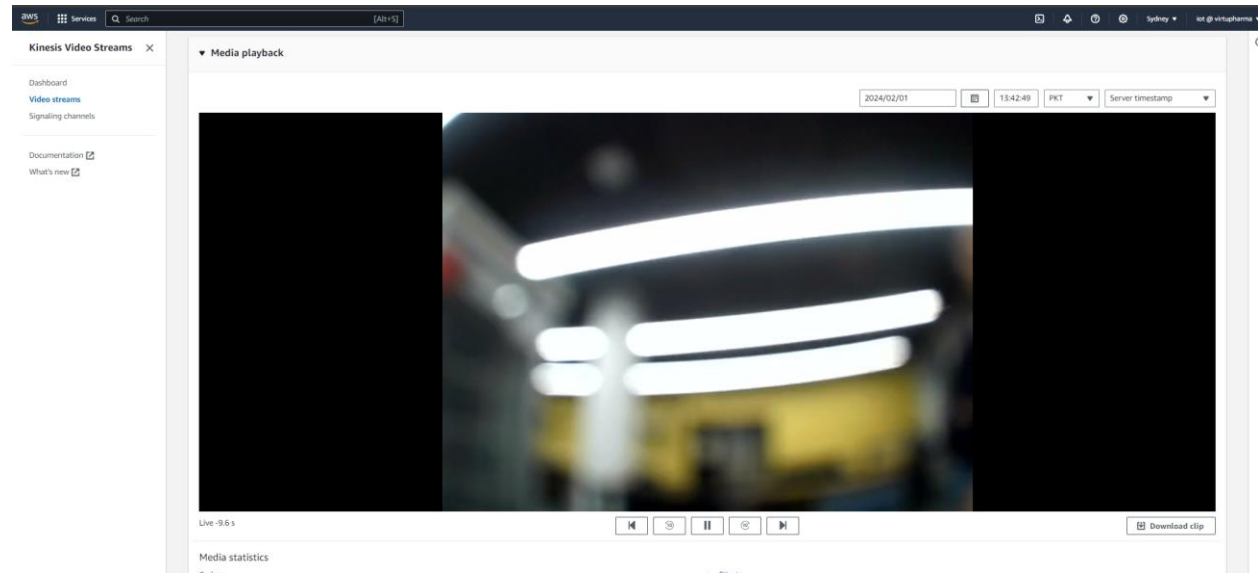


Figure 2 Kinesis Stream Playback

This section encapsulates the key aspects of the Python script, its functionality, and the anticipated visual outputs for both local and AWS cloud-based streaming components.

ESP32 Wireless Conversion for Modbus Communication:

This section outlines the integration of an ESP32 Feather Huzzah32 board into the existing Modbus communication pipeline. The initial setup involved connecting a Dixell XR70CX microcontroller to the Jetson Nano using an RS485 converter. In this improved configuration, an ESP32 board facilitates wireless communication with the Dixell microcontroller and transmits the parsed data to the Jetson Nano via Bluetooth.

Integration Overview:

ESP32 Setup:

- Utilized an ESP32 Feather Huzzah32 board equipped with an RS485 converter to interface with the Dixell XR70CX microcontroller.
- Configured the ESP32 to read data from the microcontroller via Modbus communication.

Data Parsing and Packaging:

- Implemented logic on the ESP32 to interpret Modbus data received from the Dixell XR70CX.
- Transformed the raw data into a structured JSON packet for ease of interpretation.

Wireless Data Transmission:

- Established a Bluetooth Serial connection between the ESP32 and the Jetson Nano.
- Sent the parsed data as JSON packets over the Bluetooth connection.

Jetson Nano Reception and AWS Integration:

- Modified the existing Python pipeline on the Jetson Nano to accommodate Bluetooth data reception.
- Parsed the received JSON packets and continued the pipeline to send data to AWS IoT Core.

Advantages of Wireless Integration:

- *Enhanced Flexibility:* The ESP32 introduces wireless communication, reducing the reliance on physical cables and enabling more flexible deployment options.
- *Improved Scalability:* Wireless communication simplifies the scalability of the system, allowing additional devices to be easily incorporated.

ESP32 Wireless Data Reception on Jetson Nano:

This section details the Python script used on the Jetson Nano to receive data wirelessly from an ESP32 device. The script integrates with the existing Modbus communication pipeline, facilitating the transition from a wired RS485 connection to a wireless Bluetooth connection.

Script Overview:

```
import time

import json

from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient

import csv

from bluepy.btcomm import BluetoothClient

from signal import pause

# AWS Cloud Configurations

# ... (AWS IoT Core configurations)

# MQTT Client Setup

# ... (MQTT client initialization and connection)

# Load existing data from the CSV file, if available

# ... (CSV file reading logic)

def calculate_hourly_average():

    # ... (Hourly average calculation logic)

def handle_received_data(data):

    try:

        # Parse received JSON data from ESP32

        # ... (Data parsing logic)

        # Check temperature values and set flags

        # ... (Temperature value checking logic)
```

```

# Create JSON object with temperature details
json_body = {
    # ... (JSON object creation)
}

try:
    # Publish JSON data to AWS IoT Core
    mqtt_client.publish(topic_name, json.dumps({"Payload": json_body}), 1)

    # Print success message and update CSV file
    print(f"Publish Success: Temperature Value: {temperature_celsius} °C")

    # ... (CSV file update logic)

except Exception as ex:
    # Handle AWS IoT publishing error
    print(f"Error publishing to AWS IoT: {ex}")

    # Continue updating CSV file even on AWS error
    # ... (CSV file update logic)

except json.JSONDecodeError as e:
    # Handle JSON decoding error
    print("Error decoding JSON:", e)

except Exception as e:
    # Handle unexpected errors
    print("An unexpected error occurred, Technical Support Required: ", e)

# MAC address of ESP32
esp32_mac_address = '3C:61:05:4A:DD:EA'

# Create a Bluetooth client
client = BluetoothClient(esp32_mac_address, handle_received_data)

# Wait for incoming data
pause()

```

Integration Steps:

AWS Cloud Configurations:

- Configure AWS IoT Core parameters such as endpoint, root CA path, certificate path, private key path, client ID, and topic name.

MQTT Client Setup:

- Set up an MQTT client for communication with AWS IoT Core.

CSV File Handling:

- Load existing data from a CSV file or initialize an empty list.

Data Processing and Publishing:

- Implement logic to calculate hourly averages and set flags based on temperature conditions.
- Create a JSON object containing temperature details.
- Publish the JSON data to AWS IoT Core.

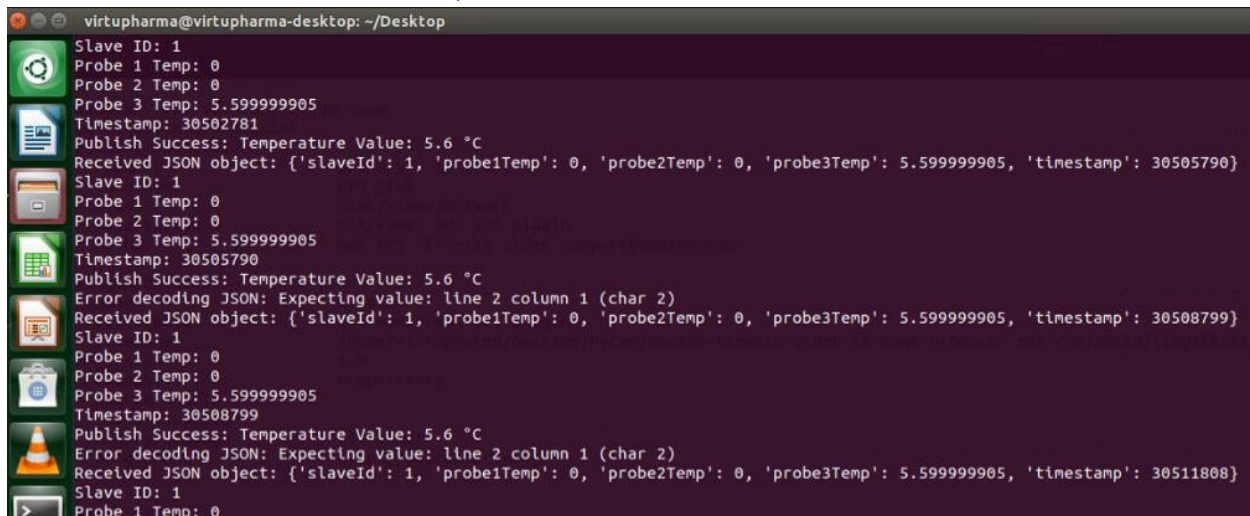
ESP32 Bluetooth Communication:

- Create a Bluetooth client on the Jetson Nano using the BluetoothClient class.
- Handle received data using the handle_received_data function.

Wireless Reception and CSV Update:

- Wait for incoming Bluetooth data using the pause function.
- Print success messages and update the CSV file with received temperature data.

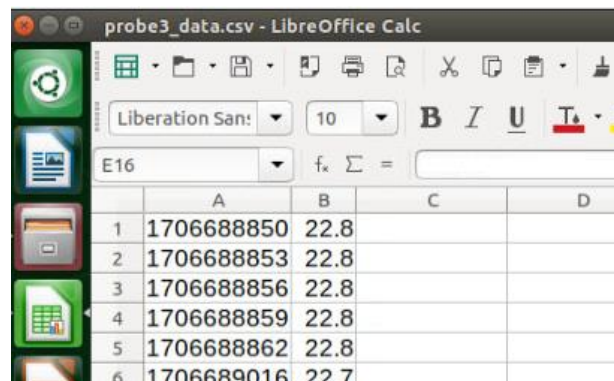
Jetson Nano Command Line Output:



```
virtupharma@virtupharma-desktop: ~/Desktop
Slave ID: 1
Probe 1 Temp: 0
Probe 2 Temp: 0
Probe 3 Temp: 5.599999905
Timestamp: 30502781
Publish Success: Temperature Value: 5.6 °C
Received JSON object: {'slaveId': 1, 'probe1Temp': 0, 'probe2Temp': 0, 'probe3Temp': 5.599999905, 'timestamp': 30505790}
Slave ID: 1
Probe 1 Temp: 0
Probe 2 Temp: 0
Probe 3 Temp: 5.599999905
Timestamp: 30505790
Publish Success: Temperature Value: 5.6 °C
Error decoding JSON: Expecting value: line 2 column 1 (char 2)
Received JSON object: {'slaveId': 1, 'probe1Temp': 0, 'probe2Temp': 0, 'probe3Temp': 5.599999905, 'timestamp': 30508799}
Slave ID: 1
Probe 1 Temp: 0
Probe 2 Temp: 0
Probe 3 Temp: 5.599999905
Timestamp: 30508799
Publish Success: Temperature Value: 5.6 °C
Error decoding JSON: Expecting value: line 2 column 1 (char 2)
Received JSON object: {'slaveId': 1, 'probe1Temp': 0, 'probe2Temp': 0, 'probe3Temp': 5.599999905, 'timestamp': 30511808}
Slave ID: 1
Probe 1 Temp: 0
```

Figure 3 Dixell Data Pipeline

Updated CSV File:



	A	B	C	D
1	1706688850	22.8		
2	1706688853	22.8		
3	1706688856	22.8		
4	1706688859	22.8		
5	1706688862	22.8		
6	1706689016	22.7		

Figure 4 CSV File Data Logging

Final Remarks:

Concluding this project, the NVIDIA Jetson Nano proved to be a reliable setup for edge computing tasks. The successful integration of USB camera streaming and the wireless transformation of a Modbus communication pipeline was done.

The collaboration between GStreamer and AWS Kinesis Video Streams facilitated smooth video streaming, laying the groundwork for real-time analytics in various applications. The parameter configuration and credential management are pivotal for optimal performance and security.

The wireless transformation of the Modbus pipeline, transitioning from a wired RS485 connection to a wireless Bluetooth setup using an ESP32, highlighted the adaptability of edge devices.

This project was tested and debugged on each stage and the final results are displayed above show the smooth working of pipelines as well as data logs.