Project Ten: IMU, GPS, and Video

Nicolas Miller

12/08/2024

1 Requirements

As specified in the assignment document, this project has been design to satisfy the following requirements:

- Capture IMU, GPS, and video data
- Transmit data to host via TCP/IP or UDP over Wifi

2 Design

I had to deconstruct my project for this deliverable. In my previous iterations, I had used mounting tape to mount the Raspberry Pi to the power supply in an enclosure beneath the drone. For this project, I opened up the Raspberry Pi and connected its SDA and SCL pins to the dedicated SDA and SCL pins on the Arduino. The IMU connections to the Arduino remained the same as the previous project, but I now had a problem of real estate.

My final implementation required the use of a pretty large basket beneath the drone. While it's not the most aesthetically pleasing solution, it provides a decent center of gravity and holds the electrical components where they need to be. If I had better foresight, I wouldn't have placed the flight controller or receiver directly on the top of the drone, but I made due with what space I had left for this implementation.

My biggest challenge for this assignment was the software design. For this deliverable, I moved away from FreeRTOS and stuck with a simple round robin with interrupts design. In short, my Arduino code defines a Wire request for the sending of data and manages two data buffers for IMU data storage and writing, which is a straightforward approach to mitigating the shared data problem of the Raspberry Pi requesting IMU data while data points are being updated.

Another interesting challenge when transmitting data over I2C involved the transmission of 16-bit integers. Because we can only transmit 8-bits at a time over I2C, I had to split the transmission, transmitting high and low bits separately, which are then combined by the script on the Raspberry Pi.

On the Raspberry Pi, I developed a simple Python script to read and log data received over I2C. This was done after configuring the Raspberry Pi to allow I2C communication. The script defines a class for IMU logging, which reads I2C data and writes it to a log file. I configured this as a system service to start on boot.

The biggest challenge was bringing all of the pieces together. In the last assignment, I utilized web socketing to provide real-time IMU data updates. I stuck with that approach in this assignment, adding GPS data to the socket updates. I spent a lot of time debugging GPS data, which turned out to be corrupted due to a deprecated Python library. Once I moved from the `gps` library to `gps3` library, everything came together.

In the end, my Flask dashboard ended up being the most straightforward approach, which uses TCP/IP under the hood to host the dashboard. Combining the straightforward Flask dashboard implementation with web socketing, I am able to provide real-time video streaming and IMU/GPS data using the Raspberry Pi as a web server.

If I had more time or was able to start everything from scratch, I would pay more mind to the economy of space on the drone. My final implementation is very cluttered. However, I did learn a lot about hardware/software design and integration, and I feel much better equipped to tackle challenges of this sort in the future.

3 Source Code

For this deliverable, I am including my Arduino code, the Raspberry Pi I2C system service code, and the Flask server and HTML. I made us of AI tools for this project, which helped me with the I2C data transmission process (splitting most and least significant bytes) and code comments.
main.cpp

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <SPI.h>

#define SLAVE_ADDRESS 0x04  // I2C address for Arduino when acting as slave
#define BNO055_SAMPLE_DELAY_MS 100  // Delay between sensor readings

// Initialize BNO055 sensor
Adafruit_BNO055 bno = Adafruit_BNO055(55);

// Structure to hold IMU orientation data
struct ImuData {
int16_t roll;
int16_t pitch;
int16_t yaw;
};

// Two buffers for double buffering
volatile ImuData buffer1;
volatile ImuData buffer2;

// Flag to track which buffer is being written to
volatile bool writing_to_first = true;

// Function declaration
void sendData();

void setup() {
// Initialize I2C communication as a slave device
Wire.begin(SLAVE_ADDRESS);
Wire.onRequest(sendData);  // Register request handler function

// Initialize serial communication for debugging
Serial.begin(9600);
Serial.println(”Starting BNO055 initialization...”);

// Initialize BNO055 sensor with error checking
```

```
if (!bno.begin()) {
    Serial.println(”Failed to initialize BNO055!”);
    Serial.println(”Check your wiring or I2C address.”);
    while (1);  // Halt if sensor initialization fails
}

Serial.println(”BNO055 initialized successfully!”);
delay(1000);  // Allow time for sensor to stabilize

// Enable external crystal for better accuracy
bno.setExtCrystalUse(true);
Serial.println(”External crystal enabled”);
}

void loop() {
sensors_event_t event;
bno.getEvent(&event);  // Get current sensor readings

// Convert floating-point angles to fixed-point integers
// Multiply by 100 to preserve 2 decimal places of precision
if (writing_to_first) {
    buffer1.roll = (int16_t)(event.orientation.z * 100);
    buffer1.pitch = (int16_t)(event.orientation.y * 100);
    buffer1.yaw = (int16_t)(event.orientation.x * 100);
} else {
    buffer2.roll = (int16_t)(event.orientation.z * 100);
    buffer2.pitch = (int16_t)(event.orientation.y * 100);
    buffer2.yaw = (int16_t)(event.orientation.x * 100);
}

// Atomic operation: switch buffers
writing_to_first = !writing_to_first;

// Print raw integer values for debugging
Serial.print(”Raw values - Roll: ”);
Serial.print(event.orientation.z * 100);
Serial.print(” Pitch: ”);
Serial.print(event.orientation.y * 100);
Serial.print(” Yaw: ”);
Serial.println(event.orientation.x * 100);

// Print original floating-point values for comparison
Serial.print(”Float values - Roll: ”);
Serial.print(event.orientation.z);
Serial.print(” Pitch: ”);
Serial.print(event.orientation.y);
Serial.print(” Yaw: ”);
Serial.println(event.orientation.x);

delay(BNO055_SAMPLE_DELAY_MS);  // Wait before next reading
}

void sendData() {
// Buffer to hold the six bytes of IMU data (2 bytes each for roll, pitch, yaw)
byte buffer[6];

// Read from the buffer that's NOT being written to
const volatile ImuData& data = writing_to_first ? buffer2 : buffer1;

// Pack 16-bit integers into bytes for transmission
// For each value, first byte is MSB (>>8), second is LSB (&0xFF)
buffer[0] = (data.roll >> 8) & 0xFF;     // Roll MSB
buffer[1] = data.roll & 0xFF;            // Roll LSB
buffer[2] = (data.pitch >> 8) & 0xFF;    // Pitch MSB
buffer[3] = data.pitch & 0xFF;           // Pitch LSB
buffer[4] = (data.yaw >> 8) & 0xFF;      // Yaw MSB
buffer[5] = data.yaw & 0xFF;             // Yaw LSB

// Send all 6 bytes over I2C
Wire.write(buffer, 6);
}
```

imu_i2c.py

```python
from smbus2 import SMBus
import time

SLAVE_ADDRESS = 0x04  # Arduino I2C slave address
LOG_FILE = "/tmp/imu_i2c_data.txt"  # Temporary file for storing IMU data

def decode_int16(msb, lsb):
    """
    Convert two bytes into a signed 16-bit integer using MSB and LSB.

    Args:
        msb (int): Most significant byte
        lsb (int): Least significant byte

    Returns:
        int: Signed 16-bit integer value (-32768 to 32767)
    """

    # Combine bytes: MSB shifted left 8 bits OR'd with LSB
    # If number is negative in two's complement
    # # Convert to negative number
    value = (msb << 8) | lsb
    if value > 32767:
        value -= 65536
    return value

class IMULogger:
    """
    Handles I2C communication with an Arduino-based IMU sensor and logs
    the data.

    The IMU sends six bytes of data representing roll, pitch, and yaw angles:
    - Bytes 0-1: Roll angle (MSB, LSB)
    - Bytes 2-3: Pitch angle (MSB, LSB)
    - Bytes 4-5: Yaw angle (MSB, LSB)

    Each angle is transmitted as a signed 16-bit integer multiplied by 100
    (to preserve two decimal places of precision).
    """

    def __init__(self):
        """
        Initialize I2C bus connection and open log file.
        """
        self.bus = SMBus(1)            # Initialize I2C bus
        self.file = open(LOG_FILE, ”a”)  # Open log file in append mode

    def read_and_log(self):
        ”””
        Read IMU data over I2C and log it to file.

        Reads 6 bytes from the Arduino, converts them to roll, pitch,
        and yaw angles, and writes them to the log file in CSV format.

        The values are divided by 100 to convert from integer to float,
        restoring the original precision.

        Format of log file: roll,pitch,yaw
        Each value is in degrees with two decimal places.

        Handles errors by printing the error message and waiting 1 second
        before the next attempt.
        ”””
        try:
            # Read 6 bytes from Arduino
            data = self.bus.read_i2c_block_data(SLAVE_ADDRESS, 0, 6)

            # Convert byte pairs to floating point angles
            roll = decode_int16(data[0], data[1]) / 100.0
            pitch = decode_int16(data[2], data[3]) / 100.0
            yaw = decode_int16(data[4], data[5]) / 100.0

            # Write angles to file in CSV format
            self.file.write(f”{roll},{pitch},{yaw}\n”)
            self.file.flush()  # Ensure data is written immediately

        except Exception as e:
            print(f”Error: {str(e)}”)
            time.sleep(1)  # Wait before retrying on error

    def close(self):
        ”””
        Clean up resources by closing the log file and I2C bus connection.
        Should be called when the logger is no longer needed.
        ”””
        self.file.close()
        self.bus.close()

def main():
    ”””
    Main program loop that continuously reads and logs IMU data.

    Creates an IMULogger instance and reads data every second.
    Handles keyboard interrupts (Ctrl+C) by closing resources.
    ”””
    logger = IMULogger()
    try:
        while True:
            logger.read_and_log()
            time.sleep(1)  # Wait 1 second between readings
    except KeyboardInterrupt:
        logger.close()  # Clean up on Ctrl+C

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

server.py

```python
from flask import Flask, render_template
from flask_socketio import SocketIO, emit
from gps3 import gps3
import os

# Initialize Flask application and SocketIO for real-time communications
app = Flask(__name__)
socketio = SocketIO(app)

# Configuration constants
IMU_BUFFER_FILE = "/tmp/imu_i2c_data.txt"

# Initialize GPS connection using GPSD daemon
try:
    gps_socket = gps3.GPSDSocket()    # Create socket connection to GPSD
    data_stream = gps3.DataStream()    # Initialize data stream object
    gps_socket.connect()               # Establish connection to GPSD daemon
    gps_socket.watch()                 # Begin watching for GPS data
    print("Connected to GPSD successfully.")

except Exception as e:
    print(f"Failed to connect to GPSD: {e}")
    gps_session = None

def read_imu_data():
    """
    Read the latest IMU data from buffer file.

    Returns:
        dict: Dictionary containing roll, pitch, and yaw values in degrees.
              Returns None for all values if data cannot be read.
    """
    try:
        if os.path.exists(IMU_BUFFER_FILE):
            with open(IMU_BUFFER_FILE, "r") as file:
                # Read all lines and get the last complete line for
                # most recent data
                lines = file.readlines()
                if lines:
                    last_line = lines[-1].strip()
                    if last_line:
                        roll, pitch, yaw = map(float, last_line.split(','))
                        print(f"Roll: {roll}, Pitch: {pitch}, Yaw: {yaw}")
                        return {'roll': roll, 'pitch': pitch, 'yaw': yaw}
    except Exception as e:
        print(f"Error reading IMU data: {e}")
    return {'roll': None, 'pitch': None, 'yaw': None}

def read_gps_data():
    """
    Read current GPS data from GPSD daemon using gps3 library.

    Returns:
        dict: Dictionary containing latitude, longitude, and altitude.
              Returns None for all values if data cannot be read.
    """
    gps_data = {
        'latitude': None,
        'longitude': None,
        'altitude': None,
    }

    for new_data in gps_socket:
        if new_data:
            data_stream.unpack(new_data)

            # TPV (Time-Position-Velocity) report contains the positioning data
            # Check if we have valid latitude data as an indicator of
            # valid GPS fix
            if data_stream.TPV['lat'] != 'n/a':
                gps_data.update({
                    'latitude': float(data_stream.TPV['lat']),
                    'longitude': float(data_stream.TPV['lon']),
                    'altitude': float(data_stream.TPV['alt'])
                })
                print(f"GPS Data - Lat: {gps_data['latitude']},
                      Lon: {gps_data['longitude']}, Alt: {gps_data['altitude']}"

            # Break after getting first valid data point
            break

    return gps_data

def emit_sensor_data():
    """
    Background task that continuously reads sensor data and emits it via
    SocketIO. Runs in an infinite loop, collecting IMU and GPS data at
    20Hz (50ms intervals). Handles errors gracefully to prevent task
    termination.
    """
    while True:
        try:
            # Collect latest sensor readings
            imu_data = read_imu_data()
            gps_data = read_gps_data()

            # Merge IMU and GPS data into single update
            combined_data = {**imu_data, **gps_data}
            socketio.emit('sensor_update', combined_data)

            # Wait 100ms before next update
            socketio.sleep(0.1)

        except Exception as e:
            print(f"Error in emit_sensor_data: {e}")
            socketio.sleep(1)

@app.route('/')
def dashboard():
    """Serve the main dashboard page."""
    return render_template('dashboard.html')

# SocketIO event handlers
@socketio.on('connect')
def handle_connect():
    """
    Handle new client connections by starting a background task
    to emit sensor data to the connected client.
    """
    socketio.start_background_task(emit_sensor_data)

if __name__ == '__main__':
    # Start the Flask application with SocketIO support
    # Debug mode enabled for development
    socketio.run(app, host='0.0.0.0', port=5000, debug=True)
```

dashboard.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Drone Dashboard</title>
<style>
    body {
        display: flex;
        flex-direction: column;
        align-items: center;
        font-family: Arial, sans-serif;
    }
    #video-container {
        margin-top: 20px;
        width: 640px;
        height: 480px;
    }
    #imu-data {
        margin-top: 20px;
        text-align: center;
    }
</style>
<script src="https://cdn.socket.io/4.5.0/socket.io.min.js"></script>
</head>
<body>
    <h1>Drone Dashboard</h1>
    <!-- Video Stream -->
    <div id="video-container">
        <iframe src="http://10.42.0.1:8081" width="640" height="480"></iframe>
    </div>

    <!-- IMU Data -->
    <div id="imu-data">
        <h3>IMU Data</h3>
        <p id="roll">Roll: Loading...</p>
        <p id="pitch">Pitch: Loading...</p>
        <p id="yaw">Yaw: Loading...</p>
    </div>

    <script src="https://cdn.socket.io/4.5.0/socket.io.min.js"></script>
    <script>
        const socket = io('http://10.42.0.1:5000');

        socket.on('imu_update', (data) => {
            document.getElementById('roll').textContent = `Roll: ${data.roll}`;
            document.getElementById('pitch').textContent = `Pitch: ${data.pitch}
            document.getElementById('yaw').textContent = `Yaw: ${data.yaw}`;
        });
    </script>
</body>
</html>
```

4 Video

I really wanted to take this one out for an actual flight, but unfortunately, my flight controller will no longer engage. I'm not sure where the problem lies, so I wasn't able to take it for a flight for this demo, which was a bit of a disappointment.

My video demonstration of the project can be found at the following link: https://youtu.be/1ZXKOplTN94

5 Sharing Statement

I will share this entire PDF submission in the sharing discussion board.