Poznań University of Technology

Faculty of Control, Robotics
and Electrical Engineering

Institute of Robotics and Machine Intelligence

Division of Control and Industrial Electronics

HTTP GET Processing and Sensor Simulation

Networks and Distributed Control Systems

Laboratory report

Ivan Iatsenko, 157566
ivan.iatsenko@student.put.poznan.pl

Maksym Babak, 157566
maksym.babak@student.put.poznan.pl

Instructor:
Adrian Wójcik, M.Sc.
Adrian.Wojcik@put.poznan.pl

11-09-2025

Contents

System Presentation

Specification Description

The system was expected to obtain all the measurements form all the SenseHAT sensors, structure the data in JSON format and send it to webpage, where a user can read the data. Hardware used: Raspberry Pi 4 1 GB RAM + SenseHAT add-on Software used: Python 3.13 Web target: Flask

System Implementation

2.1 JSON API

We use the sense_hat Python library to read hardware data and then wrap it into JSON objects, which are sent to the client via Flask endpoints.

Listing 1. Reading orientation from the sensor

```
01.     orientation = sense.get_orientation()  # pitch/roll/yaw in degrees by default
```

For example, orientation angles can be converted from degrees to radians before serialization:

Listing 2. Conversion to rads and creation of JSON structure

```
01.     # Orientation variants (radians)
02.     orientation_rad = {
03.         "roll": math.radians(orientation["roll"]),
04.         "pitch": math.radians(orientation["pitch"]),
05.         "yaw": math.radians(orientation["yaw"]),
06.     }
```

2.2 SENSOR DATA ACQUISITION

The server collects data from multiple SenseHAT sensors: temperature, humidity, pressure, and IMU (accelerometer, magnetometer, gyroscope)

Listing 3. Reading sensor data from SenseHAT

```
01.     temp = sense.get_temperature()
02.     hum = sense.get_humidity()
03.     pres = sense.get_pressure()
04.
05.     # IMU raw
06.     accel = sense.get_accelerometer_raw()
07.     mag = sense.get_compass_raw()
08.     gyro = sense.get_gyroscope_raw()
09.     orientation = sense.get_orientation()  # pitch/roll/yaw in degrees by default
10.
11.     # RGB matrix (flat ordered list of 64 [r,g,b] values)
12.     pixels = sense.get_pixels()  # returns list of 64 [r,g,b]
```

2.3 USER INPUTS + USER OUTPUTS

Below is presented function which polls get_events() for joystick in a loop and appends it to double-ended queue.

Listing 4. Joystick events polling

```
def joystick_poller():
    global _joystick_events
    while True:
        try:
            events = sense.stick.get_events()
            if events:
                with _state_lock:
                    for ev in events:
                        # convert to serializable dict
                        _joystick_events.appendleft({
                            "direction": ev.direction,
                            "action": ev.action,
                            "timestamp": ev.timestamp
                        })
            time.sleep(0.05)
        except Exception as e:
            print("Joystick poller error:", e)
            time.sleep(0.2)
```

Also we have the code snippet which enables setting of specific pixel provided by a user.

Listing 5. Setting matrix pixel form user input

```
@app.route("/api/matrix/pixel", methods=["POST"])
def api_matrix_set_pixel():

    payload = request.get_json(force=True)
    try:
        x = int(payload["x"])
        y = int(payload["y"])
        r = int(payload.get("r", 0))
        g = int(payload.get("g", 0))
        b = int(payload.get("b", 0))
    except Exception:
        return jsonify({"error": "invalid payload"}), 400

    if not (0 <= x <= 7 and 0 <= y <= 7):
        return jsonify({"error": "x,y out of range"}), 400
    for c in (r, g, b):
        if not (0 <= c <= 255):
            return jsonify({"error": "color out of range 0-255"}), 400

    sense.set_pixel(x, y, r, g, b)
    # update matrix snapshot immediately
    with _state_lock:
        _latest_matrix = sense.get_pixels()
    return jsonify({"status": "ok"})
```

2.4 GUI

In our Flask-based server, we have static file serving:

Listing 6. Static file serving

```
app = Flask(__name__, static_folder="static", static_url_path="")
```

Once we have correct URL, the function uploads index.html file to our browser:

Listing 7. html file upload

```
def index():
    return send_from_directory("static", "index.html")
```

The following properties work in the way: Receive a request(GET/POST) form the client -> read the sensor/joystick event(we use locks in order to preserve data) -> in case of POST method: validate the data, set the proper output to hardware. -> always return jsonified data.

Listing 8. Endpoints

```
@app.route("/api/sensors", methods=["GET"])
def api_sensors():

    with _state_lock:
        # return a copy so the client receives a snapshot
        payload = list(_latest_list)
    return jsonify(payload)


@app.route("/api/matrix", methods=["GET"])
def api_matrix_get():

    with _state_lock:
        matrix = list(_latest_matrix)
    return jsonify(matrix)


@app.route("/api/matrix/pixel", methods=["POST"])
def api_matrix_set_pixel():

    payload = request.get_json(force=True)
    try:
        x = int(payload["x"])
        y = int(payload["y"])
        r = int(payload.get("r", 0))
        g = int(payload.get("g", 0))
        b = int(payload.get("b", 0))
    except Exception:
        return jsonify({"error": "invalid␣payload"}), 400

    if not (0 <= x <= 7 and 0 <= y <= 7):
        return jsonify({"error": "x,y␣out␣of␣range"}), 400
    for c in (r, g, b):
        if not (0 <= c <= 255):
            return jsonify({"error": "color␣out␣of␣range␣0-255"}), 400

    sense.set_pixel(x, y, r, g, b)
    # update matrix snapshot immediately
    with _state_lock:
        _latest_matrix = sense.get_pixels()
    return jsonify({"status": "ok"})
```

2.5 DYNAMICALLY GENERATED USER INTERFACE

all the endpoints in this code(e.g. which provided below or the section above) are used to populate and dynamically update the interface.

Listing 9. Endpoint

```
"/api/sensors"
```

Also we have daemon threads here, which allow for smooth shutdown of the program once we exit the web-server(if we had daemon=False and closed the Flask server, Python would wait forever for this threads to finish)

Listing 10. Background threads

```
    t = threading.Thread(target=sampler_thread, daemon=True)
    t.start()
    j = threading.Thread(target=joystick_poller, daemon=True)
    j.start()
```

2.6 DATA SAMPLING

Background thread to periodically sample sensors and update shared state. SAMPLING_INTERVAL is set to 500 ms in order to satisfy the requirement.

Listing 11. Sampler

```
def sampler_thread():
    global _latest_list, _latest_matrix
    while True:
        try:
            new_list = read_sensors_once()
            with _state_lock:
                _latest_list = new_list
                # matrix is the item with id rgb_matrix
                for it in new_list:
                    if it["id"] == "rgb_matrix":
                        _latest_matrix = it["value"]
                        break
        except Exception as e:
            print("Sampler error:", e)
        time.sleep(SAMPLING_INTERVAL)
```

2.7 PHYSICAL UNITS

Physical units are passed inside JSON object.

Listing 12. Units inside JSON object

```
    results.append({"id": "temperature", "label": "Temperature", "value": temp, "
        unit": " C "})
    results.append({"id": "humidity", "label": "Humidity", "value": hum, "unit":
        "%"})
    results.append({"id": "pressure", "label": "Pressure", "value": pres, "unit":
         "hPa"})
```

By embedding the unit of measurement in the JSON, the client can display values unambiguously.

2.8 CONTROL RANGE

In 2.3 USER INPUTS + USER OUTPUTS one can see that requirements is satisfied. We can set any color for any pixel. This funcionality implemented here:

Listing 13. Matrix Payload

```
    payload = request.get_json(force=True)
    try:
        x = int(payload["x"])
        y = int(payload["y"])
        r = int(payload.get("r", 0))
        g = int(payload.get("g", 0))
        b = int(payload.get("b", 0))
```

Testing and Integration Results

Conclusions and Summary

Summary