**CHENG-CHIH LEE**

**29329351**

# Programming Assignment 5 – Code RED

**CHENG-CHIH LEE**

**Student ID:29329351**

**2025/12/01**

**UCI MECPS**

## #Summary

**This assignment is built upon the previous lab and creates a swarm to connect at most three ESP32's and a Raspberry Pi (as a data logger). The RPi will have a button and four LEDs (Red, Green, Yellow, White). The ESP will have an analog light sensor (photoresistor in the package).**

- **ESP 8266 LightSwarm.ino Download LightSwarm.ino(obtained from Prof. Shovic's github)**
- **Raspberry Pi LightSwarm.py Download LightSwarm.py(obtained from Prof. Shovic's github)**
  - **Python 3 version: github.com/switchdoclabs/SDL_Pi_LightSwarmLinks to an external site.**

**Each of the three ESP32's in the swarm is identical. There are no software differences and no hardware differences. They can communicate with each other by broadcasting messages to exchange sensor readings. One of them, the ESP32 with the highest reading, will become the "Master" and will forward the readings to the RPi for data logging.**
***I teared down the legacy code and fit it into FreeRTOS architecture.**

**For assignment 4:**
**Extra plotting and data log functions are added to this assignment on the RPi side.**
**External LED with PWM brightness control on the ESPs are added.**

**For assignment 5 Code RED:**
**+ESP32: An Extra LED bar is added to one ESP for indicating the brightness.**
**+RPi: An Extra LED Matrix is added to indicate dynamic brightness received.**
**A local web is developed to substitute the plot in assignment4.**

**(key words: ESP32, Raspberry Pi 5, UDP, Photocell Sensor, GPIO, FreeRTOS, threading)**

## >>>>Demo Video Link<<<<

**https://drive.google.com/file/d/1wFQlQPM65_qn-iQOzpiUm-tvtTTAlcZy/view?usp=sharing**

# #OUTLINE

# # Version History

| *Version* | *Date* | *Comment* | *Known Issue/Fix* |
|---|---|---|---|
| v01_Beta | 11/04/2025 | Beta version | |
| V01 | 11/07/2025 | Formal Release | |
| V02 | 11/18/2025 | Formal Release | With Plotting Function and ESP32 PWM LED |
| V03 | 12/01/2025 | Formal Release | LED Bar, LED Matrix, Local Web Plot |

# #Components used

1. **Photocell with a 10k pull-down resistor.**
2. **External LEDs. Need resistors (330 Ohm).**
3. **A button. Need a pull-down resistor (10k Ohm), digital input and with interrupt mode set to that pin.**
4. **LED Bar. Need 10 resistors (220 Ohm)**

# #Pin Definition for *ESP32*

| Pin Name | Definition | Usage and comments |
|---|---|---|
| LED_BUILTIN | GPIO 2 | Built in onboard LED (fixed) |
| LED_BLUE | GPIO 23 | External LED PWM pin |
| PHOTO_CELL | GPIO 34 | ADC Sampling pin for Photocell sensor |
| TEN_SEG_LED_1 | GPIO 13 | LED Bar |
| TEN_SEG_LED_2 | GPIO 12 | LED Bar |
| TEN_SEG_LED_3 | GPIO 14 | LED Bar |
| TEN_SEG_LED_4 | GPIO 27 | LED Bar |
| TEN_SEG_LED_5 | GPIO 26 | LED Bar |
| TEN_SEG_LED_6 | GPIO 25 | LED Bar |
| TEN_SEG_LED_7 | GPIO 33 | LED Bar |
| TEN_SEG_LED_8 | GPIO 32 | LED Bar |
| TEN_SEG_LED_9 | GPIO 0 | LED Bar |
| TEN_SEG_LED_10 | GPIO 4 | LED Bar |

**Table.1 Pin Definition**

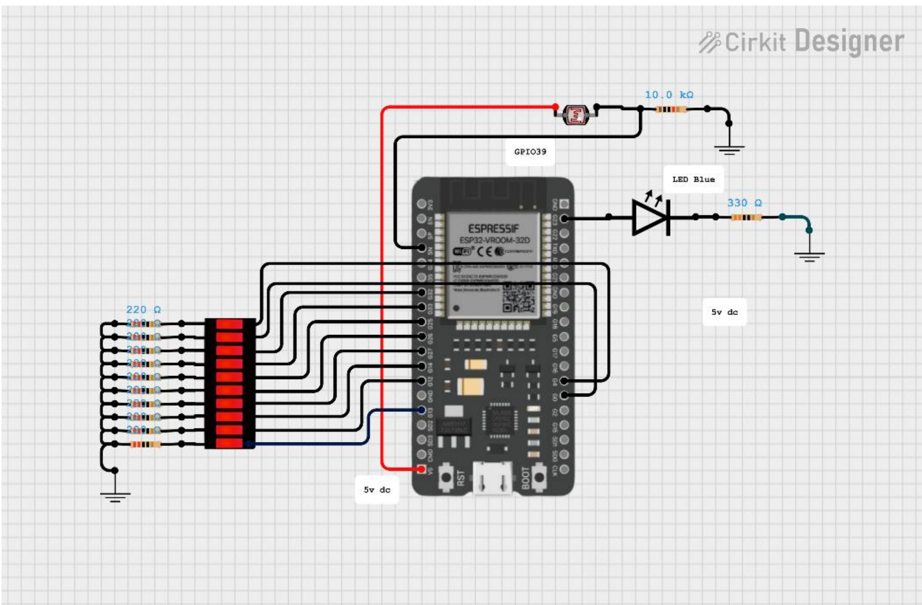



**Fig.1 Schematics**

## #Pin Definition for *Raspberry Pi*

| Pin Name | Definition | Usage and comments |
|---|---|---|
| LED_R | GPIO 26 | Pin for Red External LED |
| LED_Y | GPIO 13 | Pin for Red External YELLOW |
| LED_G | GPIO 6 | Pin for Red External GREEN |
| LED_W | GPIO 5 | Pin for Red External WHITE |
| BTN | GPIO 16 | Pin for External Push Button |
| LED_Matrix SPIMOSI | GPIO 10 | Pin for SPIMOSI |
| LED_Matrix SPICLK | GPIO 11 | Pin for SPICLK |
| LED_Matrix SPICE0 | GPIO 8 | Pin for SPICE0 |

**Table.2 Pin Definition**

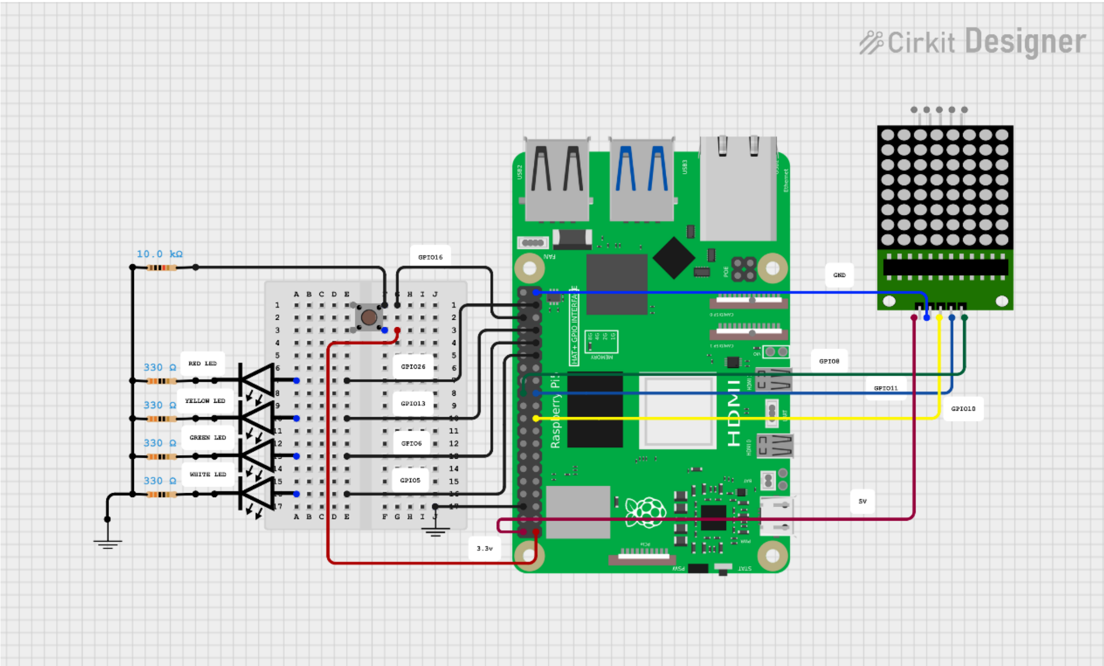
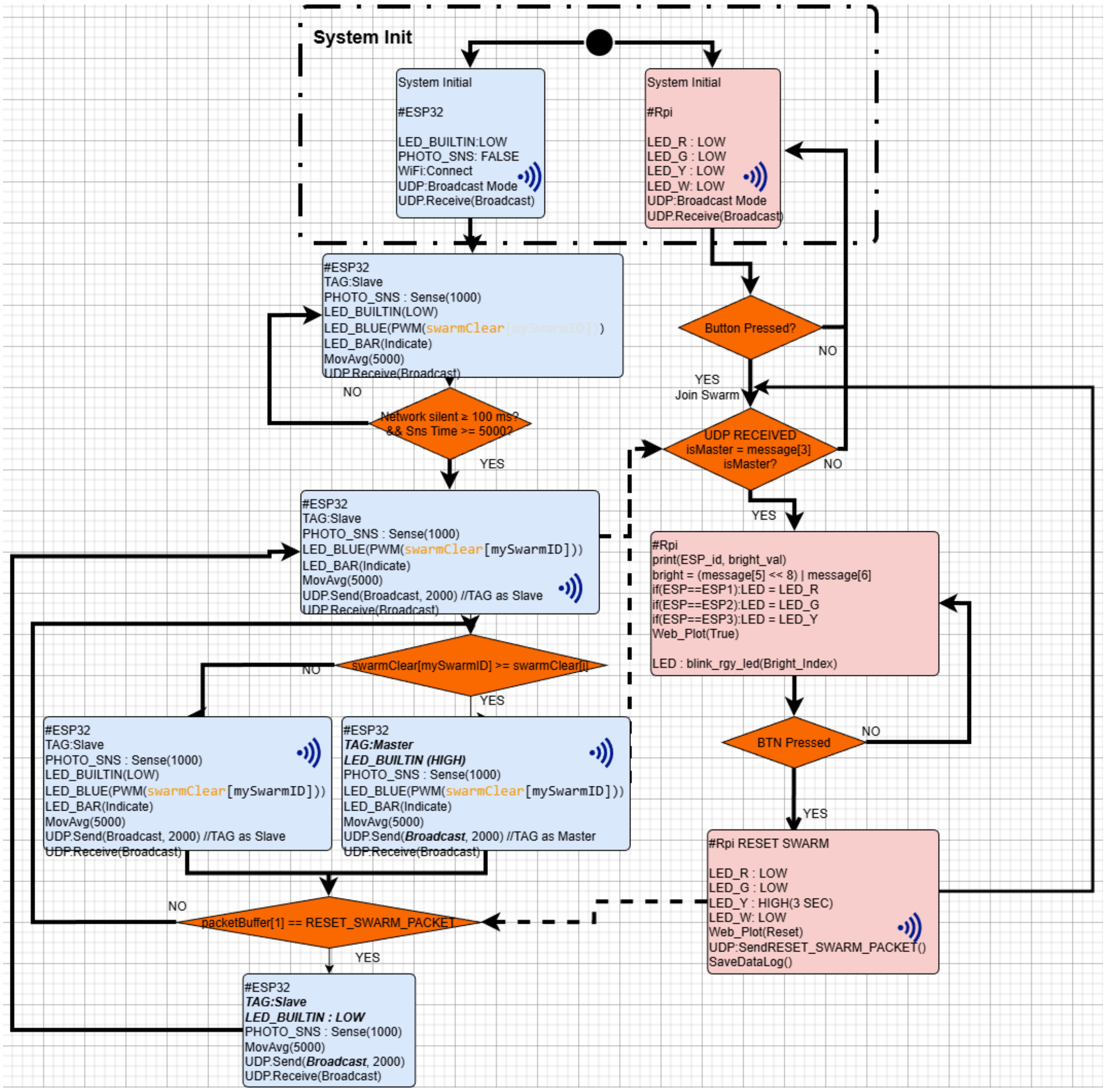


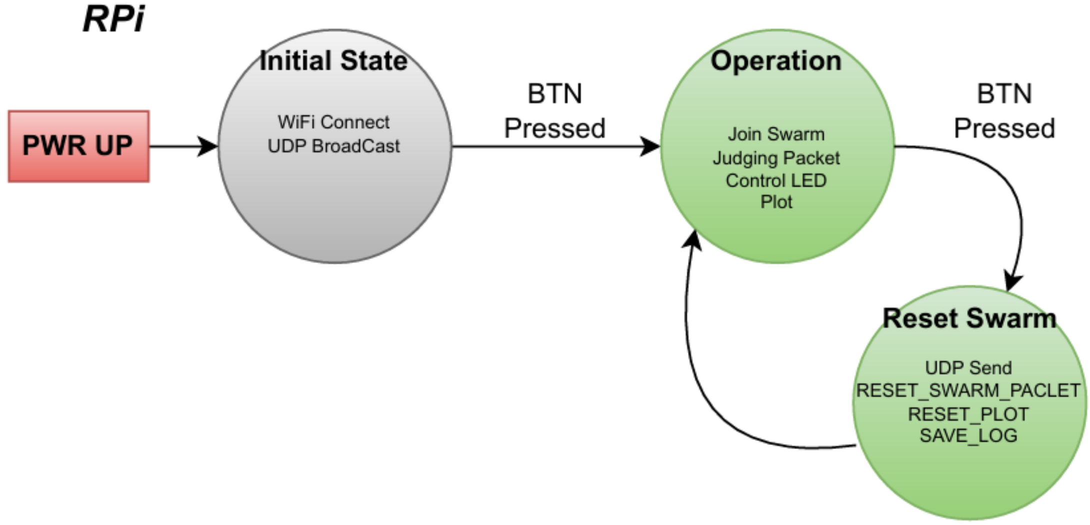

**Fig.2 Schematics**

# #Overall Flow-chart

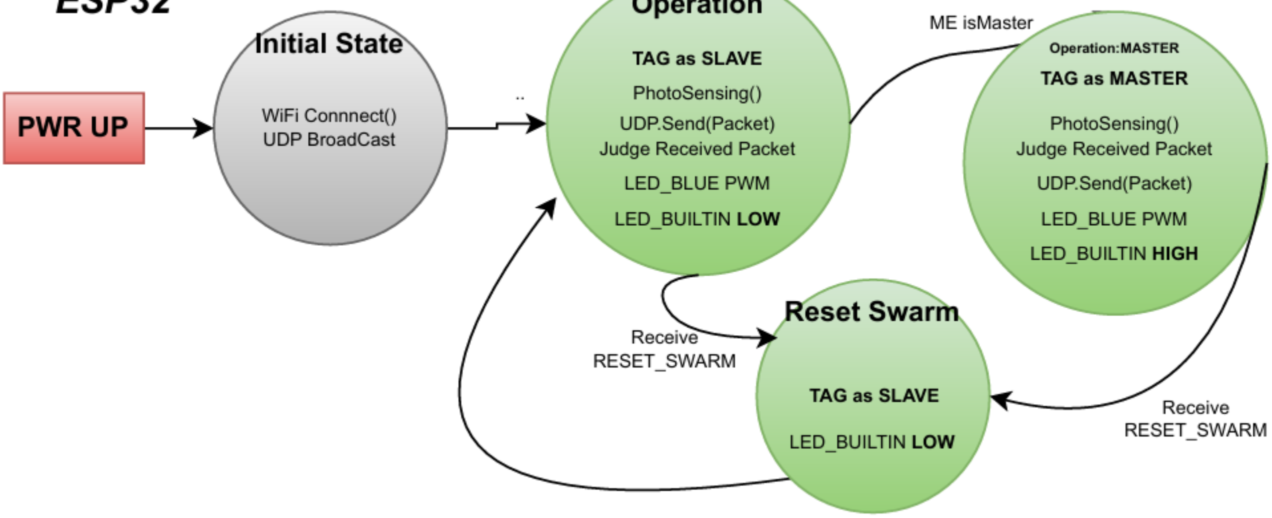


# #State Machine

## *RPi*




## *ESP32*



# #Protocols

During my development, UDP broadcasting is configured. That means all devices in the network should be able to receive all packets sent. No IP is hardcoded in any device. Port is set as 1996.

**Laptop：IP ----, Port 1996**
**RPi ：IP ----, Port 1996**
**ESP32：IP ----, Port 1996**

| Byte | Field | Value | Description |
|---|---|---|---|
| **0** | 0xF0 | Constant | **Start-of-frame marker**. Helps receivers validate the packet type. |
| **1** | LIGHT_UPDATE_PACKET (=0) | 0 | **Packet type ID** — distinguishes from reset, blink, etc. |
| **2** | localIP[3] | e.g., 123 | **Node ID** — the last octet of its IP (unique in local subnet). Used as swarm identifier. |
| **3** | masterState | 0 or 1 | **Role flag:** 1 = master, 0 = slave. |
| **4** | VERSIONNUMBER | e.g., 28 | **Firmware version** for compatibility tracking. |
| **5–6** | clearColor (high, low) | sensor value | **Clear (luminance)** 16-bit reading from the photocell (moving average). |
| **7–8** | redColor (high, low) | Not used | Placeholder for **red channel intensity**. (Unused here since no color sensor.) |
| **9–10** | greenColor (high, low) | Not used | Placeholder for green. |
| **11–12** | blueColor (high, low) | Not used | Placeholder for blue. |
| **13** | 0x0F | 0x0F | **End-of-frame marker** (symmetrical with start marker). |

## Part 1-1 Raspberry Pi WiFi setup and packet delivery

**Files: talk_v04.py, UDP_v03.py, state_machine_v05py, LightSwarm.py, plot.py, led_matrix.py, web.py**

**In this project, the LightSwarm functionality is modified/integrated into the previous "we_need_to_talk" project.** talk_v04 works as the main file, initializing some threads. UDP_v03 works as the UDP rx/tx library module. The state machine module is modified to keep only two states, including operation and reset Swarm. And the RPi is connected to my personal hotspot named IPhoneKL. WiFi related functionalities are developed in the ***"UDP_ v03.py"*** module.

### #WiFi Setup and UDP

For the UDP setup, it's set to broadcast mode, thus sending to 255.255.255.255. port is 1996.

state_machine_v03.py | talk_v03.py | UDP_v03.py | LightSwarm.py

```
#Kevin Lee 10/11/2025
#v03 Kevin Lee 11/03/2025 Modified for LightSwarm Project
import socket
print("####################################")
print("!!!!!!!!!!!!!!!!!!Hi!!!!!!>0<!!!!!!!!")
print("####################################")
#UDP_IP = input("Enter Listening IP or Enter to default 0.0.0.0: ") or "0.0.0.0"
#UDP_PORT = 8386
#Laptop_UDP_IP = input("Enter Laptop IP or Enter to default 172.20.10.4: ") or "1
#Laptop_UDP_PORT = 823
#ESP32_UDP_IP = input("Enter ESP32 IP or Enter to default 172.20.10.2: ") or "172
#ESP32_UDP_PORT = 1996
UDP_IP   = "255.255.255.255"
UDP_PORT = 1996

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)   # ? allow broadcast
sock.bind((UDP_IP, UDP_PORT))
```

In the main file (***talk_ v04.py***), I put UDP_Receive() and processPacket() and mat.show_swarm() to ***thread*** to make sure it keeps listening and driving LED matrix.
The local web is also called to run here to initialize the web page.

```python

if __name__=='__main__':

    STATE.gpio_setup()
    #init a led matrix
    mat = MAT.LED_MAT("RPi LED Mat")
    mat.spi_init(0, 0, 1000000, 0)
    time.sleep(0.2)
    mat.mat_init()
    time.sleep(0.2)

    receiver_thread = threading.Thread(target=UDP.UDP_Receive, daemon=True)
    receiver_thread.start() # Put UDP listening to thread to ensuring listening
    processPacket_thread = threading.Thread(target=LS.processPacket, daemon=True)
    processPacket_thread.start()
    ledMatrix_thread = threading.Thread(target=mat.show_swarm, daemon=True)
    ledMatrix_thread.start()

    print("#####################################################")
    print("System Up! Listening to LightSwarm Packets!!")
    print("#####################################################")

    print("=" * 60)
    print("🚀 Starting LightSwarm Web Dashboard (Real-Time Plotting)")
    print("=" * 60)

    print("\n📊 Dashboard URLs:")
    print("   Local:   http://localhost:5000")
    print("   Network: http://<raspberry-pi-ip>:5000")
    print("\n💡 Press Ctrl+C to stop")
    print("=" * 60)

    try:
        WEB.web.run(host="0.0.0.0", port=5000, debug=False, threaded=True)

    except KeyboardInterrupt:
        PLOT.plot_stop.set()
        time.sleep(0.1)
        mat.close()
        print("Exiting...")
```

This UDP module also works as a layer for state_machine and LightSwarm to exchange the LSCommand. The set functionset ***LSCommand(cmd)*** is for writing to this layer; ***get*** functions are for accessing values.

```python
def get_message_r():
    #for getting the received UDP message
    return message_r

def get_new_msg_cnt():
    #for getting the UDP message incoming status
    return new_msg_cnt

def getLSCommand():
    global LScmd
    return LScmd

def setLSCommand(cmd):
    global LScmd
    LScmd = cmd
```

For UDP **sending**, I developed ***UDP_Send()*** method to send messages to both my laptop and ESP32 for the beta version. This method will later be called by the ***state_maching*** module.

For UDP **receiving**, I developed ***UDP_Receive()*** method to receive messages. Now RPi is listening to all UDPs with this method put to thread. Upon receiving UPDs, it also updated the ***new_msg_cnt*** for checking message incoming status.

state_machine_v03.py | talk_v03.py | UDP_v03.py | LightSwarm.py

```python

message_r = "", ""
new_msg_cnt = 0
LScmd = ""

def UDP_Send(message):
    #BroadCast UDP
    sock.sendto(message, (UDP_IP, UDP_PORT))#.encode()

    print(f"BroadCasting UDP Message:'{message}' to {UDP_IP}:{UDP_PORT}")

def UDP_Receive():
    global message_r
    global new_msg_cnt

    while True:
        data, addr = sock.recvfrom(1024)
        message_r = (data, addr)  #store tuple (bytes, address)
        new_msg_cnt += 1
        #print(f"Received {len(data)} bytes from {addr}: {list(data)}")

def get_message_r():
    #for getting the received UDP message
    return message_r

def get_new_msg_cnt():
    #for getting the UDP message incoming status
    return new_msg_cnt

def getLSCommand():
    global LScmd
    return LScmd

def setLSCommand(cmd):
    global LScmd
    LScmd = cmd
```

## Part 1-2 Rpi reacts to received packets

### #State Machine

In the State machine, 2 state transitions are developed. The ***button_callback()*** is attached to external push button input interrupt for triggering state change.

```
def state_machine():
    global sys_state
    
    if sys_state == 0: #from init to operation
        m_operation()

    elif sys_state == 1:  #from norm to reset
        #photosns_stop.set()
        #blink_rgy_stop.set()
        m_reset() #to reset
        m_operation()

    elif sys_state == 2: #Reset swarm and plot
        photosns_stop.set()
        blink_rgy_stop.set()
        m_reset() #reset swarm
        m_operation()
```

## #Operation Mode *m_operation()*

This state is triggered when status flag ***sys_state* == *0*** and the **button is pressed**. The sys_state is then set to 1 indicating it's in operation mode. The phototsensing and RGY led indicating tasks will be put to threading. And the system will then remove plot stop event flag. This will un-block the plot thread in the main thread.

```python
def m_operation():
    global sys_state
    global message_r
    
    sys_state = 1
    GPIO.output(led_w, GPIO.HIGH)
    print("#################")
    print("In operation mode")
    print("#################")

    photosns_stop.clear() #clear the stop event
    photosns_thread = threading.Thread(target=photo_sns, daemon=True)
    photosns_thread.start()
    
    blink_rgy_stop.clear() #clear the stop event
    blink_RGY_thread = threading.Thread(target=blink_rgy_led, daemon=True)
    blink_RGY_thread.start()

    m_plot() #to plot state

    time.sleep(0.2)
```

A function collect_data() is developed in plot module to execute photosensor reading related for the web plotting in this assignment. It will be put to thread if plot mode is set.

```python
def m_plot():
    global sys_state
    global message_r
    global plot_enb
    
    sys_state = 2
    plot_enb = True

    PLOT.plot_stop.clear()
    plot_thread = threading.Thread(target=PLOT.collect_data, daemon=True)
    plot_thread.start()

    print("##################")
    print("Plot")
    print("##################")
```

## #Photocell value process *photo_sns()*

When photo_sns() is put to thread, with ***sys_state==1 and the thread is not stopped***. This method checks if the packet is from a master device or not, clear led indicator thread stop flag, and it also shuts down leds accordingly.

state_machine_v03.py | talk_v03.py | UDP_v03.py | LightSwarm.py

```
def photo_sns():
    global new_msg_cnt
    global pre_msg_cnt
    global led_ind
    global bright_value
    value = 0
    print("################")
    print("In photo sns now")
    print("################")
    
    while sys_state==1 and not photosns_stop.is_set():
        pre_msg_cnt = new_msg_cnt
        device_id, isMaster, value = LS.getLSMasterBright()
        #print("photo_sns Receiving Packet. device_id, isMaster, value
        new_msg_cnt = UDP.get_new_msg_cnt()
        
        if(isMaster):
            blink_rgy_stop.clear()
            bright_value = value
            print("bright_value = ",bright_value, ", value= ", value)
            if device_id is 0:
                led_ind = led_g
                #print("LED indicator is:", led_g, '\n')
                GPIO.output(led_r, GPIO.LOW)
                GPIO.output(led_y, GPIO.LOW)
            elif device_id is 1:
                led_ind = led_y
                #print("LED indicator is:", led_y, '\n')
                GPIO.output(led_r, GPIO.LOW)
                GPIO.output(led_g, GPIO.LOW)
            elif device_id is 2:
                led_ind = led_r
                #print("LED indicator is:", led_r, '\n')
                GPIO.output(led_y, GPIO.LOW)
                GPIO.output(led_g, GPIO.LOW)
                
        time.sleep(0.5)
```

## ***#Reset mode** m_reset ()*

## ***Works as reset Swarm task***

During normal operation, when the external button is pressed, the state machine will generate the photo_sns() and blink_rgy_stop() thread stopping event and sent **UDP.setLSCommand("RESETSWARM")**.

This mode will then go through stopping plot thread by calling reset_plot(), exporting log file and resetting the plot.

```
def m_reset():
    global sys_state, plot_enb
    
    sys_state = 3

    UDP.setLSCommand("RESETSWARM")#setting LS cmd to UDP layer
    print("##################")
    print("Reset Swarm")
    print("##################")
    plot_enb = False
    PLOT.ex_log() #<---log data
    PLOT.reset_plot() #<---resetting plot time and data
    
    GPIO.output(led_y, GPIO.HIGH)
    GPIO.output(led_w, GPIO.HIGH)
    GPIO.output(led_r, GPIO.LOW)
    GPIO.output(led_w, GPIO.LOW)
    GPIO.output(led_g, GPIO.LOW)
    time.sleep(3)
    GPIO.output(led_y, GPIO.LOW)
```

## Part 1-3 Rpi plot and plot reset

This assignment includes the plotting functionality. A plot module is developed and integrated into the application. There are 3 major functions in the plot module. **collect_data, reset_plot and data log exporting. Another function is get_plot_data() added for web module to fetch required data from plot module.**

### #plot

In this assignment, no plot is developed in this module. This modules now works as data collector for the local web application.

```
def collect_data():
    """Background thread: collects sensor data"""
    
    global t0, xs0, ys0, xs1, ys1, xs2, ys2, current_time
    global master_count
    
    device_id = 99
    value = 0

    print("[COLLECT] Data collection thread started")

    while not plot_stop.is_set():
        print("Collecting Data!!!!!!!!!!!!!!!-////////////////////
        
        try:
            device_id_, isMaster_, value_ = LS.getLSMasterBright()
            if isMaster_:
                device_id = device_id_
                value = value_
        except Exception as e:
            print(f"[ERROR] getLSMasterBright failed: {e}")
            time.sleep(0.1)
            continue
        
        current_timestamp = time.time() - t0
        
        if device_id == 0:
            if master_count[device_id] >= 30:
                master_count[0] = 0
                master_count[1] = 0
                master_count[2] = 0

            master_count[device_id] += 1
            xs0 = np.append(xs0, current_timestamp)
            ys0 = np.append(ys0, value)
            current_time = xs0[-1]
            
            if current_time > WINDOW:
                mask = xs0 >= (current_time - WINDOW)
                xs0 = xs0[mask]
                ys0 = ys0[mask]
```

```python
        elif device_id == 1:
            if master_count[device_id] >= 30:
                master_count[0] = 0
                master_count[1] = 0
                master_count[2] = 0

            master_count[device_id] += 1
            xs1 = np.append(xs1, current_timestamp)
            ys1 = np.append(ys1, value)
            current_time = xs1[-1]

            if current_time > WINDOW:
                mask = xs1 >= (current_time - WINDOW)
                xs1 = xs1[mask]
                ys1 = ys1[mask]

        elif device_id == 2:
            if master_count[device_id] >= 30:
                master_count[0] = 0
                master_count[1] = 0
                master_count[2] = 0

            master_count[device_id] += 1
            xs2 = np.append(xs2, current_timestamp)
            ys2 = np.append(ys2, value)
            current_time = xs2[-1]

            if current_time > WINDOW:
                mask = xs2 >= (current_time - WINDOW)
                xs2 = xs2[mask]
                ys2 = ys2[mask]

        print("master_count = ", master_count[0], master_count[1], master_count[2])

        time.sleep(1)
```

## #reset_plot()

When the reset_plot is called, the data for bar chart data will be cleared, the time axis will be set to zero and setting a reset flag for the plot() thread to clear the data chart.

```python
def reset_plot():
    """Reset all plot data"""
    global t0, xs0, ys0, xs1, ys1, xs2, ys2, current_time
    global master_count, reset_counter

    reset_counter+=1

    plot_stop.set() #<---stopping plot thread

    t0 = time.time()
    print("Resetting Data!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    time.sleep(5)
    xs0 = np.array([])
    ys0 = np.array([])
    xs1 = np.array([])
    ys1 = np.array([])
    xs2 = np.array([])
    ys2 = np.array([])
    master_count = [0, 0, 0]
    t0 = time.time()
    current_time = 0

    print(f"[RESET] Counter: {reset_counter}")
```

```python
def plot():

    global t0, xs0, ys0, xs1, ys1, xs2, ys2, current_time, line0, line1, line2, master_count, fig
    device_id = 99
    value = 0

    ax1.set_xlabel("Time (s)")
    ax1.set_ylabel("PhotoCell Reading")
    ax1.set_title("LightSwarm Brightness")
    ax1.set_ylim(0, 5000)  # adjust based on brightness range
    ax2.set_xlabel("Device ID")
    ax2.set_ylabel("Accumulative Master Count")
    ax2.set_title("Device Master Chart")
    ani = animation.FuncAnimation(fig, update_plot, interval=1000)
    plt.show(block=False)

    while not plot_stop.is_set():

        if plot_reset_flag.is_set(): #reset plot
            print("Resetting Plot!!\n")
            xs0.clear(); ys0.clear()
            xs1.clear(); ys1.clear()
            xs2.clear(); ys2.clear()

            line0.set_xdata([]); line0.set_ydata([])
            line1.set_xdata([]); line1.set_ydata([])
            line2.set_xdata([]); line2.set_ydata([])

            ax1.set_xlim(0, WINDOW)

            plot_reset_flag.clear()
```

# #ex_log ()

When the ex_log is executed this function export the data in the previous 30 secs to a .txt file.

```python
def ex_log():
    global xs0, ys0, xs1, ys1, xs2, ys2, current_time, master_count

    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

    os.makedirs("log", exist_ok=True)
    with open(f"log/{timestamp}.txt", "w") as f:
        f.write("=== LightSwarm Data Log ===\n")
        f.write(f"Timestamp: {timestamp}\n\n")

        #Log master counter
        f.write("Master Count per Device:\n")
        f.write(f"Device 0: {master_count[0]}\n")
        f.write(f"Device 1: {master_count[1]}\n")
        f.write(f"Device 2: {master_count[2]}\n\n")

        #Log arrays
        f.write("=== Device 0 Data (xs0, ys0) ===\n")
        for t, v in zip(xs0, ys0):
            f.write(f"{t:.2f}, {v}\n")

        f.write("\n=== Device 1 Data (xs1, ys1) ===\n")
        for t, v in zip(xs1, ys1):
            f.write(f"{t:.2f}, {v}\n")

        f.write("\n=== Device 2 Data (xs2, ys2) ===\n")
        for t, v in zip(xs2, ys2):
            f.write(f"{t:.2f}, {v}\n")

    print(f"[LOG] Exported data → {timestamp}.txt")
```

*~/Desktop/ECPS216/swarm2/log/20251118_154955.txt - Mousepad

File Edit Search View Document Help

```
=== LightSwarm Data Log ===
Timestamp: 20251118_154955

Master Count per Device:
Device 0: 6
Device 1: 5
Device 2: 0

=== Device 0 Data (xs0, ys0) ===
31.56, 2368
32.58, 2368
33.60, 2368
34.61, 2368
35.62, 2368
36.64, 2354
37.65, 2354
38.66, 2354
39.68, 2308
40.69, 2308
41.70, 2308
42.72, 2308
43.73, 2308
44.75, 2252
72.17, 2559
73.19, 2559
74.20, 2555
```

# #web

***I used ZOTGPT to developed this part.***

In this module, flask library is used to realize the local web application. Object "web" is instantiated at the top of this module.

```python
from flask import Flask, render_template_string, jsonify
import plot as PLOT
import state_machine_v05 as STATE

web = Flask(__name__)
```

ZotGPT developed a HTML that worked pretty well for the web so I just kept it.

```python
HTML_TEMPLATE = """
<!DOCTYPE html>
<html>
<head>
  <title>LightSwarm Real-Time Dashboard</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    * {
      margin: 0;
      padding: 0;
      box-sizing: border-box;
    }
    
    body {
      font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
      background: linear-gradient(135deg, #0d0d0d 0%, #1a1a1a 100%);
      color: white;
      min-height: 100vh;
      padding: 20px;
    }
    
    .container {
      max-width: 1400px;
      margin: 0 auto;
    }
    
    header {
      text-align: center;
      margin-bottom: 30px;
    }
    
    h1 {
```

5 endpoints were developed. They are home, get_data, reset_plot, export_log and status.

```python
@web.route("/")
def home():
    """Main dashboard page"""
    return render_template_string(HTML_TEMPLATE)

@web.route("/data")
def get_data():
    """API endpoint: Get current plot data"""
    data = PLOT.get_plot_data()
    return jsonify(data)

@web.route("/reset", methods=["POST"])
def reset_plot():
    """API endpoint: Reset plot"""
    STATE.state_machine()
    return jsonify({
        "status": "success",
        "message": "Plot reset successfully!"
    })

@web.route("/export", methods=["POST"])
def export_log():
    """API endpoint: Export log"""
    try:
        PLOT.ex_log()
        return jsonify({
            "status": "success",
            "message": "Log exported successfully!"
        })
    except Exception as e:
        return jsonify({
            "status": "error",
            "message": f"Export failed: {str(e)}"
        }), 500

@web.route("/status")
def status():
    """API endpoint: System status"""
    data = PLOT.get_plot_data()
    return jsonify({
        "status": "running",
        "master_count": data['master_count'],
        "window_size": PLOT.WINDOW,
        "data_points": {
            "device0": len(data['time0']),
            "device1": len(data['time1']),
            "device2": len(data['time2'])
        }
    })
```

This the what the web looks like:

## -Part 1-4 Rpi LED Matrix

In this assignment, a LED Matrix is added to indicate the dynamic sensor reading from ESP32 master device. I used the LED 8*8 matrix from the kit. It comes with MAX7219CNG controller. SPI protocol is used to communicate with it.

Basic usage is to send it (register, data). For just controlling the matrix, the register 1-9 stands for matrix rows 1-9; data is as easy as 8 bits 1/0 and converted to hex. For example, to control on row 2, led 2, we need to send (2, 0100 0000(hex)).

I developed a class for controlling the matrix. Initialization requires setting spi0 module, go through the several settings and I set all leds to LOW.

The MAT_VAL array at the top is ready for controlling LED on status from none to all 8 leds.

```
MAT_VAL = [0x00, 0x80, 0xC0, 0xE0, 0xF0, 0xF8, 0xFC, 0xFE, 0xFF]
DATA = [0, 0, 0, 0, 0, 0, 0, 0, 0]
raw_value = 0

class LED_MAT:
    def __init__(self, name):
        # Constructor (runs when object is created)
        self.name = name
        self.status = False

    def spi_init(self, bus, device, max_spd, mode):

        self.spi = spidev.SpiDev()
        self.spi.open(bus, device)  # bus 0, device 0 (spidev0.0)
        self.spi.max_speed_hz = max_spd  # 1 MHz
        self.spi.mode = mode
        self.status = True

    def mat_init(self):
        # MAX7219 init
        self.show(0x09, 0x00)  # no decode
        self.show(0x0A, 0x08)  # brightness (0x00-0x0F)
        self.show(0x0B, 0x07)  # scan all 8 rows
        self.show(0x0C, 0x01)  # exit shutdown mode
        self.show(0x0F, 0x00)  # display test off

        for row in range(1, 9):
            self.show(row, 0x00)
        time.sleep(0.5)
```

To make the LED matrix show dynamic and continuous data, I decided to put the received value to a data queue.

So the show_swarm() function will push new data into the data_queue and call show() function to send all data in the updated data_queue through spi.

show_swarm() is also put to thread in the system init status.

```
    def data_queue(self, data):
        DATA[8] = DATA[7]
        DATA[7] = DATA[6]
        DATA[6] = DATA[5]
        DATA[5] = DATA[4]
        DATA[4] = DATA[3]
        DATA[3] = DATA[2]
        DATA[2] = DATA[1]
        DATA[1] = DATA[0]
        DATA[0] = data

    def show(self, reg, data):
        if self.status is True:
            self.spi.xfer2([reg, data])
        else:
            print("LED Matrix Not Yet Init YO~~")

    def close(self):
        self.spi.xfer2([0x0C, 0x00])

    def show_swarm(self):
        global raw_value

        while True:
            device_id_, isMaster_, value_ = LS.getLSMasterBright()
            if(isMaster_):
                device_id = device_id_
                raw_value = value_
                raw_value /= 511 #to 0 - 8

            if raw_value > 8:
                raw_value = 8
            if raw_value < 0:
                raw_value = 0
            raw_value = int(raw_value)
            value = MAT_VAL[raw_value] #get hex
            self.data_queue(value) #push data to DATA[]
            for i in range (0, 8):
                self.show(i+1, DATA[i])

            time.sleep(1)
```

# Part 2-1 ESP WiFi setup and packet delivery

ESP is connected to my hotspot. It is also developed to UDP broadcasting.

LightSwarm.ino

```
char ssid[] = "iPhoneKL";  //  your network SSID (name)
char pass[] = '[redacted]; // your network password

unsigned int localPort = 1996; // local port to listen for UDP packets
IPAddress serverAddress = IPAddress(255, 255, 255, 255); // default no IP Address
```

```
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, pass);

  // initialize Swarm Address - we start out as swarmID of 0

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");

  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());

  Serial.println("Starting UDP");

  udp.begin(localPort);       // bind & listen on port
  //udp.setBroadcast(true);     // allow broadcast TX
```

**Swarm data array is initialized here.**

```
  // initialize light sensor and arrays
  int i;
  for (i = 0; i < SWARMSIZE; i++)
  {
    swarmAddresses[i] = 0;
    swarmClear[i] = 0;
    swarmTimeStamp[i] = -1;
  }
  swarmClear[mySwarmID] = 0;
  swarmTimeStamp[mySwarmID] = 1;   // I am always in time to myself
  clearColor = swarmClear[mySwarmID];
  swarmVersion[mySwarmID] = VERSIONNUMBER;
  swarmState[mySwarmID] = masterState;
  Serial.print("clearColor =");
  Serial.println(clearColor);
  // set SwarmID based on IP address
  localIP = WiFi.localIP();
  swarmAddresses[0] =  localIP[3];
  mySwarmID = 0;
  Serial.print("MySwarmID=");
  Serial.println(mySwarmID);
```

## #Tasks

After WiFi connection, six tasks are created. **UDP receiving Task** for UDP receiving, **Task_PhotoSns** for Photocell sensing, **Task_UDP_Send** for sending UDP packet. **Task_Swarm** for Swarm logic implementation. **Task_LED_Blink** for controlling Built-In led blinking behavior. **Task_Monitor** for checking stack watermark.

```
/*Mutesx to protect shared swarm data field*/
swarmMutex = xSemaphoreCreateMutex();
/*Task Creation*/
xTaskCreatePinnedToCore(Task_PhotoSns, "Task_PhotoSns", 4096, NULL, 1, &xHandle_Task_PhotoSns, 1);
xTaskCreatePinnedToCore(Task_UDP_Receive, "Task_UDP_Receive", 4096, NULL, 2, &xHandle_Task_UDP_Receive, 1);
xTaskCreatePinnedToCore(Task_UDP_Send, "Task_UDP_Send", 4096, NULL, 1, &xHandle_Task_UDP_Send, 1);
xTaskCreatePinnedToCore(Task_Swarm, "Task_Swarm", 4096, NULL, 1, &xHandle_Task_Swarm, 1);
xTaskCreatePinnedToCore(Task_LED_Blink, "Task_LED_Blink", 4096, NULL, 1, &xHandle_Task_LED_Blink, 1);
xTaskCreatePinnedToCore(Task_Monitor, "Task_Monitor", 4096, NULL, 1, NULL, 1);
```

## #UDP Receiving *UDP_Receive(void)*

In the UDP_Receive() function, UDP incoming messages are processed by calling the Packet_Helper function. There's a mutex set to protect the incoming packet.

```
void Task_UDP_Receive(void *pvParameters){

  for(;;){
    int cb = udp.parsePacket();
    if (cb) {
      udp.read(packetBuffer, PACKET_SIZE);
      xSemaphoreTake(swarmMutex, portMAX_DELAY);
// update swarm arrays and clearColor
      Incoming_Packet_Helper(packetBuffer);
      xSemaphoreGive(swarmMutex);
    }
    vTaskDelay(100 / portTICK_PERIOD_MS);
  }
}
```

## #UDP Sending *UDP_Send(const char* msg)*

In the UDP_Send() function, broadcastARandomUpdatePacket() is called and sendLightUpdatePacket() is called accordingly.

```
void Task_UDP_Send(void *pvParameters){
  for(;;){
    vTaskDelay(1);                  // let RX run first
    broadcastARandomUpdatePacket();
    vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}
```

```
void broadcastARandomUpdatePacket()
{
  int sendToLightSwarm = 255;
  int randomDelay;
  randomDelay = random(0, MAXDELAY);
  Serial.print("Delay = ");
  Serial.print(randomDelay);
  Serial.print("ms : ");

  vTaskDelay(randomDelay / portTICK_PERIOD_MS);

  IPAddress sendSwarmAddress(255, 255, 255, sendToLightSwarm); // my Swarm Address
  sendLightUpdatePacket(sendSwarmAddress);
}
```

# Part 2-2 ESP correctly reacts to received packets

## #Task_Swarm

The Task_Swarm is for checking the main logic for Light Swarm by calling checkAndSetIfMaster().

```
void Task_Swarm(void *pvParameters){
  for(;;){
    // Check to see if I am master!
    checkAndSetIfMaster();
    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}
```

```
void checkAndSetIfMaster()
{
  int i;
  int howLongAgo;
  for (i = 0; i < SWARMSIZE; i++)
  {

#ifdef DEBUG
    Serial.print("swarmClear[");
    Serial.print(i);
    Serial.print("] = ");
    Serial.print(swarmClear[i]);
    Serial.print("  swarmTimeStamp[");
    Serial.print(i);
    Serial.print("] = ");
    Serial.println(swarmTimeStamp[i]);
#endif
    Serial.print("#");
    Serial.print(i);
    Serial.print("/");
    Serial.print(swarmState[i]);
    Serial.print("/");
    Serial.print(swarmVersion[i]);
    Serial.print(":");
    // age data
    howLongAgo = millis() - swarmTimeStamp[i]
```

# #Photocell sensing function

The Photo_Sns(void) function is developed to execute sensing with analog reading. A moving average function MovAvg(uint16_t input, uint16_t *buffer) is used to derive 5 sec sliding window average.

LightSwarm.ino

```

int Photo_Sns(void){
  int val = 0u;

  val = analogRead(PHOTO_SNS_PIN);
  val = MovAvg(val, &PHOTO_READINGS[0]);/*Moving average*/
  data_cnt+=1;
  sprintf(photo_sns_buffer, "%d", val);  /*convert int → text*/
  Serial.print("Photo Sns Read=");
  Serial.print(photo_sns_buffer);
  Serial.println(" ");
  return val;
}

uint16_t MovAvg(uint16_t input, uint16_t *buffer){
  /*Buffer with 5 elements*/
  float avg = 0.0f;

  buffer[4] = buffer[3];
  buffer[3] = buffer[2];
  buffer[2] = buffer[1];
  buffer[1] = buffer[0]; /*pop out old element and push new*/
  buffer[0] = input;

  avg = (buffer[0]+buffer[1]+buffer[2]+buffer[3]+buffer[4])/5;

  return avg;
}
```

# #Task_LED_Blink function

This function is now modified to set PWM duty to the external Blue LED pin to indicate the photo sensor reading. The reading is 0~4095, so I set the duty resolution to 2^12.

```
pinMode(LED_BUILTIN, OUTPUT);
analogWriteResolution(LED_BLE, 12); //0-4095
analogWriteFrequency(LED_BLE, 5000);  // 0 kHz global PWM frequency
analogWrite(LED_BLE, 0);   //duty
/*Start*/
```

```
void Task_LED_Blink(void *pvParameters){
  for(;;){
    Serial.print("Bright Value = ");
    Serial.print(swarmClear[mySwarmID]);
    Serial.print('\n');
    analogWrite(LED_BLE, swarmClear[mySwarmID]);
    //vTaskDelay(BLNK_MS / portTICK_PERIOD_MS);
    vTaskDelay(50 / portTICK_PERIOD_MS);
  }
}
```

# #Task_LED_Bar

This function was developed to control the 10 segment LED bar. The LED bar is as easy as ten leds. Controlling HIGH/LOW of GPIO is enough. To make it easier, I store the leds to an array, so the execution function only needs to run a loop to scan through leds needed to be turned on and off.

This function is also created as a FreeRTOS task with cycle time 200ms.

```
uint8_t led_seg[] = {
    TEN_SEG_LED_1,
    TEN_SEG_LED_2,
    TEN_SEG_LED_3,
    TEN_SEG_LED_4,
    TEN_SEG_LED_5,
    TEN_SEG_LED_6,
    TEN_SEG_LED_7,
    TEN_SEG_LED_8,
    TEN_SEG_LED_9,
    TEN_SEG_LED_10,
};
```

```
void Task_LED_Bar(void *pvParameters){
  uint8_t i = 0;
  uint8_t k = 0;
  uint8_t j = 0;
  for(;;){
    i = clearColor/409;

    for(j = i; j<11; j++){
      digitalWrite(led_seg[j], LOW);
    }
    for(k = 0; k<i; k++){
      digitalWrite(led_seg[k], HIGH);
    }

    vTaskDelay(200 / portTICK_PERIOD_MS);
  }
}
```