Software Development Practice 1

Instructor: RSP <rawat.s@eng.kmutnb.ac.th>

MicroPython-ESP32 for IoT Applications

Objectives

- Learn how to write MicroPython code for ESP32 / ESP32-S3 boards using the latest version of MicroPython firmware and its API.

Expected Learning Outcomes

After completing all tasks, students will be able to:

- Write MicroPython code to utilize the hardware resources of the ESP32 microcontroller.
- Develop software projects that utilize MicroPython-ESP32 to implement simple IoT applications.

Introduction

MicroPython is a C implementation of the Python 3 runtime environment, targeted at various microcontroller devices and boards. Espressif SoCs, such as the ESP32 and ESP32-S3, are two popular microcontrollers suitable for IoT applications. In this assignment, we will explore some features of the network stack, including WiFi, BLE, HTTP, and WebSocket protocols, which are supported by MicroPython on the ESP32.

Task 1

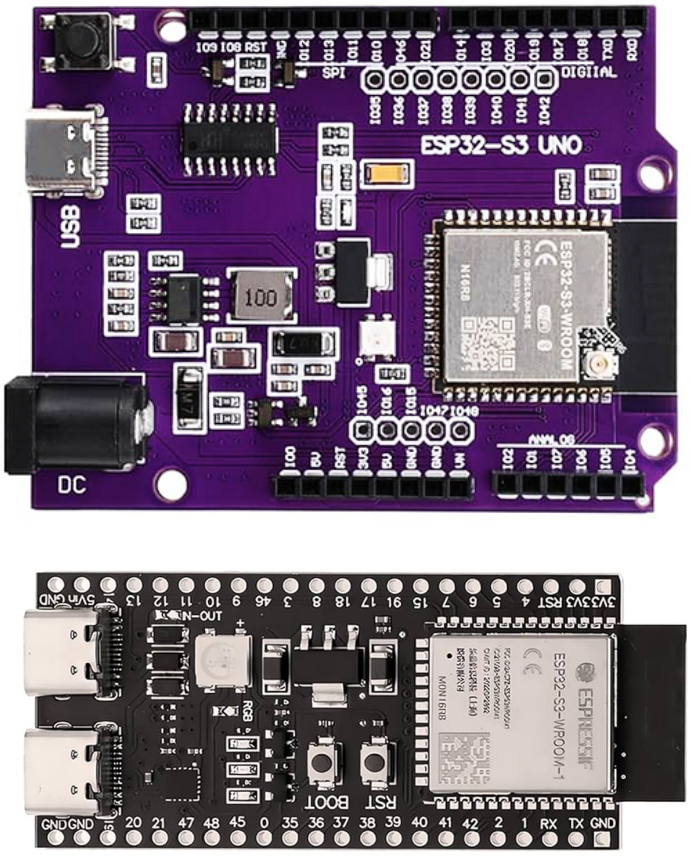
Some ESP32-S3 boards include an onboard RGB LED (WS2812B-based). We will begin with writing MicroPython code to set or change the color of the RGB LED. MicroPython provides a built-in library called `neopixel` for this purpose.

1.1) Write MicroPython code for the ESP32 or ESP32-S3 that changes the color of an RGB LED (WS2812B, also known as NeoPixel) randomly every 1 second. The color should be selected from a predefined list containing at least 6 different colors. Use MicroPython’s built-in 'neopixel' library for controlling the WS2812B LED.

1.2) Modify the code from Task 1.1 so that when the user presses a button, the RGB LED stops changing colors. Pressing the button again should resume the random color changes.

Note:

- For task 1.2, do not use the time.sleep() or time.sleep_ms() function.
- You can use the onboard BOOT button (active-low), which is connected to the GPIO-0 of the ESP32 / ESP32-S3 chip.
- You can also use ESP32 boards with external WS2812B modules.



Examples of ESP32-S3 boards with the ESP32-S3-WROOM-1 module
(ESP32-S3 N16R8 chip, 16 MB Flash, and on-chip 8 MB Octal PSRAM)

Task 2

In this task, we will explore how to write MicroPython code for the ESP32 / ESP32-S3 to connect to a WiFi access point and serve as a simple embedded web server with WebSocket support. The main objective is to implement a simple web app that allows the user to set or change the onboard RGB LED by utilizing the WebSocket protocol.

2.1) Test the MicroPython code on the ESP32 / ESP32-S3 board. This demonstrates how to use the ESP32 / ESP32-S3 as a simple HTTP server with WebSocket support.

Note:

- Save the WiFi SSID and password for the WiFi access point in the secrets.json file (on the Flash file system).
- Use a web browser (e.g., Chrome, Edge) to connect the web server and test the web application.

2.2) Rewrite the code from Task 2.1 so that:

- While the ESP32 / ESP32-S3 is connecting to Wi-Fi, the WS2812B RGB LED blinks.
- Once Wi-Fi is connected, the RGB LED turns green.

2.3) Revise the server-side and client-side code so that when the user clicks a button on the web page, the RGB LED color changes randomly.

2.4) Write MicroPython code (both server-side and client-side) so that when the user presses the button on the ESP32 / ESP32-S3 board, the visual indicator on the web page toggles its state accordingly.

Note about WebSocket Handshake Protocol

A client (e.g., a web browser) sends a normal **HTTP/1.1 request** to the server, asking to upgrade the connection to WebSocket. This includes headers like the example below:

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: <random Base64-encoded string>
Sec-WebSocket-Version: 13
```

If the server supports WebSockets, it responds with **HTTP 101 Switching Protocols**. The server takes the client's Sec-WebSocket-Key, appends the magic GUID 258EAFA5-E914-47DA-95CA-C5AB0DC85B11, hashes it with SHA-1, and encodes the result with Base64.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: <calculated Base64 string>
```

Extra Task

Evaluate at least two of following open-source projects for WebSocket-based application using MicroPython-ESP32:

1. https://pypi.org/project/micropython-async-websocket-client/
2. https://github.com/miguelgrinberg/microdot
3. https://github.com/AlecGMcNamara/Micropython-ESP32-WebServer-with-sockets

Describe how to install, explore, and evaluate these software projects.
Explain which of these projects (possibly with some modifications) can be deployed successfully on ESP32 or ESP32-S3.

MicroPython code for Task 2

```
import network
import socket
import uasyncio as asyncio
import sys
import struct
import ubinascii
import hashlib
import ujson

with open("secrets.json", "r") as f:
    secrets = ujson.load(f)

WIFI_SSID = secrets["WIFI_SSID"]
WIFI_PASS = secrets["WIFI_PASS"]

HTML_CONTENT = """\
HTTP/1.0 200 OK\r
Content-Type: text/html\r
\r
<!DOCTYPE html>
<html>
  <body>
    <h1>MicroPython WebSocket Demo</h1>
    <button onclick="count+=1; ws.send('Hello ESP32! #' + count)">Send</button>
    <pre id="log"></pre>
    <script>
      let count = 0;
      const log = document.getElementById('log');
      const ws = new WebSocket('ws://' + location.host);
      ws.onmessage = e => { log.textContent += "ESP32: " + e.data + "\\n"; };
    </script>
  </body>
</html>
"""

async def connect_wifi():
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    if not wlan.isconnected():
        wlan.connect(WIFI_SSID, WIFI_PASS)
        while not wlan.isconnected():
            await asyncio.sleep(0.5)  # non-blocking wait
    print("Connected:", wlan.ifconfig())
    return wlan

async def websocket_handler(reader, writer, headers):
    magic_guid = b"258EAFA5-E914-47DA-95CA-C5AB0DC85B11"
    # Extract the 'Sec-WebSocket-Key' header from the request.
    key = ""
    for h in headers:
        if b"Sec-WebSocket-Key" in h:
            # Extract the key for WebSocket hankshake
            key = h.split(b": ")[1].strip()
            break

    hash_value = hashlib.sha1(key + magic_guid).digest()
    sec_ws_accept = ubinascii.b2a_base64( hash_value ).strip()
    await writer.awrite(
        b"HTTP/1.1 101 Switching Protocols\r\n"
        b"Upgrade: websocket\r\n"
        b"Connection: Upgrade\r\n"
```

```
            b"Sec-WebSocket-Accept: " + sec_ws_accept + b"\r\n\r\n"
        )
        print("WebSocket connected…")

        try:
            while True:
                # Caution: for payloads shorter than 126 bytes
                data = await reader.read(2)
                if not data or data[0] != 0x81:
                    break
                length = data[1] & 0x7f
                mask = await reader.read(4)
                payload = bytearray(await reader.read(length))
                for i in range(length):
                    payload[i] ^= mask[i % 4]
                print( payload.decode('utf-8') )

                # Echo back: We send the received message back
                # Make a frame to send back
                frame = bytearray([0x81, len(payload)]) + payload
                await writer.awrite(frame)
        except Exception as e:
            print("WS error:", e)
        finally:
            await writer.aclose()
            print("WebSocket disconnected")

    async def http_handler(writer):
        await writer.awrite(HTML_CONTENT)
        await writer.aclose()

    async def handle_http_client(reader, writer):
        try:
            # Read HTTP headers
            headers = []
            while True:
                line = await reader.readline()
                if line == b"\r\n" or line == b"":
                    break
                headers.append(line)

            # WebSocket upgrade request?
            if any(b"Upgrade: websocket" in h for h in headers):
                await websocket_handler(reader, writer, headers)
            else:
                await http_handler(writer)
        except Exception as e:
            print("Client error:", e)
            await writer.aclose()

    async def main():
        await connect_wifi()
        await asyncio.start_server(handle_http_client, "0.0.0.0", 80)
        print("Server running...")
        while True:
            await asyncio.sleep(1000)

    try:
        asyncio.run(main())
    finally:
        asyncio.new_event_loop()
```

Task 3

In this task, we will explore how to use the **Bluetooth / BLE feature** of the ESP32 / ESP32-S3, including the associated MicroPython libraries. We begin with a sample MicroPython code (written in an OOP style) to implement a simple **BLE scanner**. Next, we will explore how to implement a web app that utilizes the **Web Bluetooth protocol** to interact with the ESP32 / ESP32-S3 using BLE services (Nordic BLE UART service).

3.1) Use the provided MicroPython code to scan BLE devices.

- Revise the code to save the BLE scan results in a .json file on the flash file system, avoiding duplicate entries of BLE devices.
- Revise the code to display more information about a specific BLE MAC address.

3.2) To implement a BLE peripheral on the ESP32 / ESP32-S3, you need the following files from Micropython GitHub: https://github.com/micropython/micropython/.

- /examples/bluetooth/ble_advertising.py
- /examples/bluetooth/ble_uart_peripheral.py

The ble_uart_peripheral.py file, which requires the ble_advertising.py file, provides BLE UART services on the ESP32 / ESP32-S3. Initially, it starts by advertising the GATT service and waiting for incoming connections. You can use a mobile app, such as **Nordic nRF Connect**, to scan and connect BLE devices and services if they are in discoverable mode.

3.3) Write Python code using the Flask framework to implement a web server and serve **index.html**, which represents a simple web app.

- Test your web app by scanning for BLE devices and connecting to the ESP32 / ESP32-S3 device.

Extra Task

- Suggest ideas for utilizing the WiFi / BLE features of the ESP32 / ESP32-S3 to implement a web-based game with physical interaction.
- Explain how your system would function.

File: ble_scam.py

```
import uasyncio as asyncio
import ubluetooth
import ubinascii

class AsyncBLEScanner:

    def __init__(self):
        self.ble = ubluetooth.BLE()
        self.ble.active(True)
        self.ble.irq(self.bt_irq)
        self.devices = {}

    def bt_irq(self, event, data):
        # Event 5 = _IRQ_SCAN_RESULT
        if event == 5:
            addr_type, addr, adv_type, rssi, adv_data = data
            mac = ":".join("%02X" % b for b in reversed(addr))
            if mac not in self.devices:
                self.devices[mac] = {"rssi": rssi}
                #print(f"MAC: {mac}, RSSI: {rssi}")
        elif event == 6:
            print("Scan done.\n")

    async def scan_loop(self, duration_ms=3000, interval_s=5):
        while True:
            self.devices = {}
            print("Start async BLE scan...")
            self.ble.gap_scan(duration_ms, 30000, 30000)
            await asyncio.sleep(duration_ms / 1000)
            print("Scan complete. Found devices:")
            for mac, info in self.devices.items():
                print( f"MAC: {mac}, RSSI: {info['rssi']}" )
            await asyncio.sleep(interval_s)

async def main():
    scanner = AsyncBLEScanner()
    await scanner.scan_loop()

try:
    asyncio.run(main())
finally:
    asyncio.new_event_loop()
```

File: index.html

```
<!DOCTYPE html>
<html>
<body>
<h1>ESP32 UART via Web Bluetooth</h1>
<button id="connect">Connect</button>
<button id="send">Send Message</button>
<pre id="log"></pre>

<script>
let device, server, txChar, rxChar;
const log = document.getElementById("log");

document.getElementById("connect").onclick = async () => {
    try {
        device = await navigator.bluetooth.requestDevice({
            acceptAllDevices: true,
            optionalServices: ['6e400001-b5a3-f393-e0a9-e50e24dcca9e']
        });
        device.addEventListener('gattserverdisconnected', () => {
            log.textContent += "Device disconnected\n";
        });

        server = await device.gatt.connect();

        const service = await
           server.getPrimaryService('6e400001-b5a3-f393-e0a9-e50e24dcca9e');
        txChar = await
           service.getCharacteristic('6e400003-b5a3-f393-e0a9-e50e24dcca9e');
        rxChar = await
           service.getCharacteristic('6e400002-b5a3-f393-e0a9-e50e24dcca9e');

        await txChar.startNotifications();

        txChar.addEventListener('characteristicvaluechanged', e => {
            let value = new TextDecoder().decode(e.target.value.buffer);
            log.textContent += "Received: " + value + "\n";
        });
        log.textContent += "Connected!\n";
    } catch(e) {
        log.textContent += "Error: " + e + "\n";
    }
};

document.getElementById("send").onclick = async () => {
    if (!rxChar) return;
    let msg = "Hello ESP32!";
    try {
        await rxChar.writeValue(new TextEncoder().encode(msg));
        log.textContent += "Sent: " + msg + "\n";
    } catch(e) {
        log.textContent += "Send error: " + e + "\n";
    }
};
</script>

</body>
</html>
```