TECNICATURA SUPERIOR EN

Telecomunicaciones

Electrónica Microcontrolada

Optimización 2: Comunicación y Preprocesamiento

ISPC INSTITUTO SUPERIOR POLITÉCNICO CÓRDOBA

Dirección General de EDUCACIÓN TÉCNICA Y FORMACIÓN PROFESIONAL — Ministerio de EDUCACIÓN — CÓRDOBA entre todos

Profesor: Gonzalo Vera

Alumno: Raúl Jara

Librerías Utilizadas

Librería WiFi.h

La librería **WiFi.h** que estamos utilizando en tu proyecto es la estándar de Arduino para controlar las conexiones Wi-Fi del ESP32. Proporciona funciones necesarias para manejar la conectividad a redes, cambiar entre diferentes modos de operación (estación, punto de acceso), y monitorear el estado de la conexión. Aquí te explico cómo funciona, sus métodos principales y cómo se relaciona con los modos de trabajo del ESP32.

Modos de Trabajo del ESP32

El **ESP32** tiene tres modos de funcionamiento principales en cuanto a conectividad Wi-Fi:

1. **Modo Estación (STA)**: El ESP32 se conecta a una red Wi-Fi existente, actuando como un dispositivo cliente, similar a cómo lo hace tu computadora o tu teléfono. Este es el modo que estás utilizando en tu práctica.
2. **Modo Punto de Acceso (AP)**: El ESP32 actúa como un router, permitiendo que otros dispositivos se conecten a su red.

3. **Modo Dual (AP + STA)**: El ESP32 puede actuar simultáneamente como estación y punto de acceso, conectándose a una red Wi-Fi mientras permite que otros dispositivos se conecten a él.

Principales Métodos de la Librería WiFi.h

1. WiFi.begin(ssid, password)

- Inicia la conexión Wi-Fi en modo estación con las credenciales proporcionadas (SSID y contraseña).
- Mientras el ESP32 está intentando conectarse, la función no bloquea el código, permitiendo que otras partes del programa se ejecuten.

2. WiFi.status()

- Devuelve el estado actual de la conexión Wi-Fi. Los estados comunes incluyen:
    - WL_CONNECTED: Significa que el ESP32 está conectado a la red Wi-Fi.
    - WL_IDLE_STATUS: El Wi-Fi está en modo inactivo, esperando a que se realice una acción.
    - WL_NO_SSID_AVAIL: No se encuentra el SSID.
    - WL_CONNECT_FAILED: Fallo en la conexión, generalmente debido a credenciales incorrectas.
    - WL_DISCONNECTED: Desconectado de la red.

3. WiFi.localIP()

- Devuelve la dirección IP asignada al ESP32 por la red Wi-Fi una vez que se establece la conexión.

4. WiFi.disconnect()

- Desconecta el ESP32 de la red Wi-Fi actual.

5. WiFi.reconnect()

- Intenta reconectar el ESP32 a la red Wi-Fi. Útil cuando la conexión se pierde y se desea reconectar sin volver a configurar el SSID y la contraseña.

6. WiFi.mode(WIFI_STA)

- Establece el modo de trabajo del ESP32. En este caso, el parámetro WIFI_STA indica que el ESP32 se comportará como un cliente en una red Wi-Fi (modo estación).

7. WiFi.softAP(ssid, password)

- Utilizado en modo **Punto de Acceso (AP)**. Permite que el ESP32 cree su propia red Wi-Fi, donde ssid y password son las credenciales de la red creada.

8. WiFi.softAPIP()

- Devuelve la dirección IP asignada al ESP32 cuando está funcionando en modo Punto de Acceso.

Control de Conexión

En una conexión básica , estamos utilizando **WiFi.status()** para monitorear constantemente el estado de la conexión Wi-Fi. Si detecta que el estado no es **WL_CONNECTED**, intentamos reconectar automáticamente con **WiFi.reconnect()**.

Esta lógica te permite:

- **Monitorear el estado**: El ESP32 está constantemente revisando si está conectado a la red. Si no lo está, imprime un mensaje indicando que ha perdido la conexión.
- **Reconectar automáticamente**: En caso de desconexión, el código intenta reconectar utilizando **WiFi.reconnect()**. Si la reconexión es exitosa, se imprime un mensaje de confirmación.
- **Tiempo de espera**: Se añade un tiempo de espera entre cada intento de reconexión (delay(5000)), lo que evita que el ESP32 intente reconectarse de manera continua sin descanso.

Resumen del Flujo de Conexión en el Modo Estación

1. Se configura el ESP32 para conectarse a la red Wi-Fi (modo estación).
2. El ESP32 intenta conectarse utilizando las credenciales proporcionadas.
3. Se monitorea continuamente el estado de la conexión.
4. En caso de pérdida de la conexión, el ESP32 intentará reconectar automáticamente después de un tiempo de espera.

Este proceso te asegura que el ESP32 pueda mantenerse conectado de manera fiable a la red Wi-Fi y que sea capaz de reconectar en caso de que la red falle.

Librería WebServer.h

La librería **WebServer.h** (no es **WebAServer.h**) permite al **ESP32** actuar como un servidor web simple utilizando **Wi-Fi**. Es parte del framework de **Arduino** y proporciona métodos y clases para crear servidores web, manejar solicitudes HTTP, y servir páginas o recursos estáticos.

M**étodos principales** y cómo configurar el ESP32 como un servidor web básico:

1. Configuración básica del ESP32 como servidor web

Configurar el ESP32 para que se conecte a una red Wi-Fi y actúe como un servidor web. Esto se logra mediante la combinación de las librerías WiFi.h y WebServer.h.

1.1. Instalación de la librería

La librería WebServer.h viene preinstalada en el framework de **Arduino** para ESP32, por lo que no necesitas agregarla manualmente. Solo necesitas incluirla en tu código:

```
#include <WiFi.h>

#include <WebServer.h>
```

1.2. Configuración del ESP32 para conectarse a Wi-Fi

Antes de que el ESP32 pueda actuar como servidor, debe conectarse a una red Wi-Fi. Aquí se definen las credenciales de la red:

```
const char* ssid = "Tu_SSID";

const char* password = "Tu_Contraseña";
```

1.3. Crear una instancia del servidor web

Puedes crear una instancia del servidor web en un puerto específico (por defecto, el puerto 80 para HTTP):

```
WebServer server(80);  // El servidor escucha en el puerto 80
```

2. Métodos principales de WebServer.h

A continuación, se describen los métodos más utilizados de la librería:

2.1. Comenzar el servidor

El método begin() inicia el servidor web. Debes llamarlo después de establecer la conexión Wi-Fi.

```
server.begin();
```

2.2. Manejo de rutas y solicitudes

Para manejar solicitudes HTTP, usas el método on(), que define rutas (endpoints) específicas para que el servidor las maneje.

- **on()**: Define un manejador para una ruta específica.
    - Sintaxis: server.on(path, HTTP_METHOD, handler_function)

Ejemplo:

```
server.on("/", HTTP_GET, []() {
  server.send(200, "text/plain", "Bienvenido al servidor web del ESP32");
});
```

En este caso, la ruta / devuelve un mensaje de texto plano cuando se accede a la página principal del servidor.

2.3. Responder a solicitudes

- **send()**: Envía una respuesta al cliente que hizo la solicitud.
    - Sintaxis: server.send(status_code, content_type, response_body)

Ejemplo:

server.send(200, "text/html", "<h1>Hola desde el ESP32!</h1>");

Esto envía una respuesta HTML con el código de estado 200 (OK).

2.4. Método para procesar solicitudes entrantes

El método handleClient() es esencial para mantener el servidor en funcionamiento. Se debe llamar repetidamente dentro del bucle principal del código:

server.handleClient();

Este método procesa cualquier solicitud entrante que el servidor reciba.

2.5. Manejador de página no encontrada (404)

Puedes definir una página personalizada para cuando un cliente solicita una ruta no existente:

```
server.onNotFound([]() {
  server.send(404, "text/plain", "Página no encontrada");
});
```

Librería HTTPClient.h

La librería **HTTPClient.h** es una de las herramientas más útiles para gestionar solicitudes HTTP desde el **ESP32**. Te permite enviar solicitudes HTTP GET, POST, PUT, DELETE, entre otras, a servidores remotos y recibir respuestas, lo que es esencial en proyectos que requieren interacción con APIs o servicios web.

1. Configuración básica

Primero, debes incluir la librería HTTPClient.h en tu código. Es parte del framework de **Arduino para ESP32**, así que no necesitas instalarla manualmente si estás usando PlatformIO o el IDE de Arduino.

```
#include <WiFi.h>
#include <HTTPClient.h>
```

2. Métodos principales de HTTPClient.h

A continuación, se presentan los métodos y funciones clave que ofrece la librería **HTTPClient.h**:

2.1. begin()

Este método inicia la conexión con el servidor al que deseas hacer la solicitud. Necesitas proporcionar la URL o URI a la que quieres acceder.

- Sintaxis:

```
http.begin(url);
```

- Ejemplo:

```
HTTPClient http;

http.begin("http://jsonplaceholder.typicode.com/posts");
```

Puedes proporcionar el protocolo http o https dependiendo del servidor.

2.2. addHeader()

Este método te permite agregar encabezados HTTP a tu solicitud, lo cual es útil para incluir tokens de autenticación o tipos de contenido específicos.

- Sintaxis:

```
http.addHeader("Nombre-Encabezado", "Valor");
```

- Ejemplo:

```
http.addHeader("Content-Type", "application/json");
```

2.3. GET()

Este método envía una solicitud **GET** al servidor. Es útil cuando deseas recuperar datos de un servidor.

- Sintaxis:

```
int httpCode = http.GET();
```

- Ejemplo:

```
int httpCode = http.GET();  // Envía la solicitud GET

if (httpCode > 0) {

  String payload = http.getString();  // Obtiene el cuerpo de la respuesta

  Serial.println(payload);

} else {

  Serial.println("Error en la solicitud GET");

}
```

2.4. POST()

Este método envía una solicitud **POST** al servidor, típicamente para enviar datos o realizar alguna operación en el servidor.

- Sintaxis:

```
int httpCode = http.POST(cuerpo_de_datos);
```

- Ejemplo:

```
int httpCode = http.POST("{\"titulo\":\"Nuevo post\", \"cuerpo\":\"Contenido\"}");
if (httpCode > 0) {
  String payload = http.getString();  // Obtiene la respuesta del servidor
  Serial.println(payload);
} else {
  Serial.println("Error en la solicitud POST");
}
```

2.5. PUT()

Este método envía una solicitud **PUT** para actualizar datos en un servidor.

- Sintaxis:

```
int httpCode = http.PUT(cuerpo_de_datos);
```

- Ejemplo:

```
int httpCode = http.PUT("{\"titulo\":\"Actualización\", \"cuerpo\":\"Nuevo contenido\"}");
```

2.6. DELETE()

Este método envía una solicitud **DELETE** al servidor para eliminar un recurso en particular.

- Sintaxis:

```
int httpCode = http.sendRequest("DELETE");
```

- Ejemplo:

```
int httpCode = http.sendRequest("DELETE");
if (httpCode > 0) {
  Serial.println("Recurso eliminado con éxito");
} else {
  Serial.println("Error al eliminar el recurso");
}
```

2.7. getString()

Este método te permite obtener el cuerpo de la respuesta del servidor en formato de texto. Es útil después de una solicitud **GET** o **POST** para procesar los datos devueltos.

- Sintaxis:

```
String respuesta = http.getString();
```

2.8. getStream()

Si estás esperando recibir un archivo grande u otro tipo de datos en forma de stream, este método te permite acceder a los datos recibidos en un flujo de bytes.

- Sintaxis:

```
WiFiClient *stream = http.getStreamPtr();
```

3. Manejo de códigos de estado HTTP

Cuando realizas solicitudes HTTP, el servidor responde con un código de estado HTTP que indica si la operación fue exitosa o no. Puedes obtener este código con la variable httpCode.

Ejemplos de códigos de estado:

- **200**: OK (Solicitud exitosa)
- **201**: Created (Recurso creado con éxito)
- **404**: Not Found (El recurso solicitado no existe)
- **500**: Internal Server Error (Error en el servidor)

```
if (httpCode > 0) {
  if (httpCode == 200) {
    Serial.println("Solicitud exitosa");
  }
} else {
  Serial.printf("Error en la solicitud: %s\n", http.errorToString(httpCode).c_str());
}
```

4. Cierre de la conexión con end()

Después de que la solicitud HTTP ha sido procesada, es importante liberar los recursos llamando a **end()**:

- Sintaxis:

```
http.end();
```

- Ejemplo completo:

```
HTTPClient http;

http.begin("http://jsonplaceholder.typicode.com/posts/1");

int httpCode = http.GET();


if (httpCode > 0) {
  String payload = http.getString();
```

```
  Serial.println(payload);
} else {
  Serial.println("Error en la solicitud");
}

http.end();  // Liberar recursos
```

5. Código completo: Solicitud GET con manejo de errores

Aquí tienes un ejemplo completo de cómo realizar una solicitud GET y manejar la respuesta:

```
#include <WiFi.h>
#include <HTTPClient.h>

const char* ssid = "Tu_SSID";
const char* password = "Tu_Contraseña";

void setup() {
  Serial.begin(115200);

  // Conectar a Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Conectando a Wi-Fi...");
  }
  Serial.println("Conectado a Wi-Fi");

  // Realizar solicitud GET
  if ((WiFi.status() == WL_CONNECTED)) {
    HTTPClient http;
    http.begin("http://jsonplaceholder.typicode.com/posts/1");  // URL del servidor
    int httpCode = http.GET();  // Realizar la solicitud GET
```

```
    // Si la solicitud fue exitosa
    if (httpCode > 0) {
      String payload = http.getString();  // Obtener la respuesta
      Serial.println(payload);
    } else {
      Serial.println("Error en la solicitud GET");
    }
    http.end();  // Finalizar la conexión
  }
}
void loop() {
  // Nada aquí
}
```

6. Consideraciones para solicitudes HTTPS

Si deseas hacer solicitudes a un servidor que use **HTTPS**, deberás asegurarte de que el certificado sea válido, o usar WiFiClientSecure. La lógica de HTTPClient es la misma para solicitudes HTTPS, pero en lugar de usar http.begin(url), deberías pasar un cliente seguro:

```
WiFiClientSecure client;
client.setInsecure();  // Usar esto si no deseas verificar el certificado
http.begin(client, "https://servidor-seguro.com");
```

7. Conclusión

La librería **HTTPClient.h** es extremadamente útil para interactuar con servidores web desde un ESP32. Proporciona métodos para realizar diversas solicitudes HTTP (GET, POST, PUT, DELETE) y manejar las respuestas, lo cual es crucial en proyectos que requieren conectarse a servicios externos o APIs.