custom c file for header

```c
#include "hl_wifi.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "esp_log.h"
#include "esp_system.h"
#include "nvs_flash.h"
#include "esp_netif.h"
#include "lwip/err.h"
#include "lwip/sys.h"
#include "lwip/sockets.h" // Already included for sockaddr_in_t
#include <string.h>
#include <arpa/inet.h>
#include <errno.h>

static const char *TAG = "hl_wifi";

static void event_handler(void* arg, esp_event_base_t event_base,
                          int32_t event_id, void* event_data)
{
    if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START) {
        esp_wifi_connect();
    } else if (event_base == WIFI_EVENT && event_id ==
WIFI_EVENT_STA_DISCONNECTED) {
        esp_wifi_connect();
        ESP_LOGI(TAG, "retry to connect to the AP");
    } else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t* event = (ip_event_got_ip_t*) event_data;
        ESP_LOGI(TAG, "got ip:" IPSTR, IP2STR(&event->ip_info.ip));

        connect_callback_t callback = (connect_callback_t)arg;
        if (callback) {
            callback();
        }
    }
}

void hl_wifi_init(connect_callback_t callback)
{
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    esp_netif_create_default_wifi_sta();

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    esp_event_handler_instance_t instance_any_id;
    esp_event_handler_instance_t instance_got_ip;
    ESP_ERROR_CHECK(esp_event_handler_instance_register(WIFI_EVENT,
                                                        ESP_EVENT_ANY_ID,
                                                        &event_handler,
```

```
                                                        NULL,
                                                        &instance_any_id));
    ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_EVENT,

IP_EVENT_STA_GOT_IP,
                                                        &event_handler,
                                                        (void*)callback,
                                                        &instance_got_ip));

    wifi_config_t wifi_config = {
        .sta = {
            .ssid = CONFIG_ESP_WIFI_SSID,
            .password = CONFIG_ESP_WIFI_PASSWORD,
            .threshold.authmode = WIFI_AUTH_WPA2_PSK,
            .pmf_cfg = {
                .capable = true,
                .required = false
            },
        },
    };
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA) );
    ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &wifi_config) );
    ESP_ERROR_CHECK(esp_wifi_start() );

    ESP_LOGI(TAG, "wifi_init_sta finished.");
}

// New wrapper function to create a sockaddr_in_t
sockaddr_in_t hl_wifi_make_addr(char* ip_str, uint16_t port)
{
    sockaddr_in_t addr; // Declare a variable addr of type sockaddr_in_t
    memset(&addr, 0, sizeof(addr)); // Initialize to zeros

    addr.sin_addr.s_addr = inet_addr(ip_str); // Pass the char function
parameter through the inet_addr function and assign the result to
addr.sin_addr.s_addr
    addr.sin_family = AF_INET; // Assign AF_INET to addr.sin_family
    addr.sin_port = htons(port); // Pass the uint16_t function parameter
through the htons function and assign the result to addr.sin_port

    return addr; // Return addr
}

int hl_wifi_tcp_connect(sockaddr_in_t addr)
{
    int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP); // Call socket
    if (sock < 0) { // Handle negative value of sock
        ESP_LOGE(TAG, "Failed to create socket: errno %d", errno);
        return -1;
    }

    int err = connect(sock, (struct sockaddr *)&addr, sizeof(struct
sockaddr_in)); // Call connect
    if (err != 0) { // Handle non-zero value of err
```

```
        ESP_LOGE(TAG, "Socket connect failed: errno %d", errno);
        close(sock); // Close the socket on failure
        return -1;
    }

    ESP_LOGI(TAG, "Successfully connected socket.");
    return sock; // Return sock
}

void hl_wifi_tcp_tx(int sock, void* buffer, uint16_t length)
{
    uint16_t offset = 0; // Declare an offset variable initialized to zero

    while (offset < length) { // Loop while offset < length
        uint16_t remainder = length - offset; // Calculate remainder

        // Call send with sock, (char*)buffer + offset, remainder, and 0
        int sent_bytes = send(sock, (char*)buffer + offset, remainder, 0);

        if (sent_bytes < 0) { // Handle negative return value (error)
            ESP_LOGE(TAG, "Error sending data: errno %d", errno);
            // Depending on the error, you might want to break, retry, or
close the socket.
            // For now, we'll just log and exit the function.
            return;
        } else if (sent_bytes == 0) { // Should generally not happen for
TCP unless connection is closed
            ESP_LOGE(TAG, "Send returned 0 bytes, connection might be
closed.");
            return;
        } else {
            offset += sent_bytes; // Update offset with the number of bytes
actually transmitted
        }
    }
    ESP_LOGI(TAG, "Successfully sent %u bytes.", length);
}
```

and this is the custom high level header made for this

```
#ifndef HL_WIFI_H
#define HL_WIFI_H

#include "esp_wifi.h"
#include "esp_event.h"
#include "lwip/sockets.h" // Required for struct sockaddr_in

// New type definition for connect callback
typedef void (*connect_callback_t)(void);

// New type definition for sockaddr_in
typedef struct sockaddr_in sockaddr_in_t;
```

```c
/**
 * Initialize WiFi in station mode and connect to configured network
 * @param callback A function pointer to be called when the device gets an
IP address.
 */
void hl_wifi_init(connect_callback_t callback);

/**
 * @brief Creates a sockaddr_in_t structure from an IP address string and
port.
 * @param ip_str The IP address as a string (e.g., "192.168.1.100").
 * @param port The port number.
 * @return A populated sockaddr_in_t structure.
 */
sockaddr_in_t hl_wifi_make_addr(char* ip_str, uint16_t port);

/**
 * @brief Establishes a TCP connection to the specified address.
 * @param addr The target address structure.
 * @return The socket descriptor on success, or -1 on failure.
 */
 int hl_wifi_tcp_connect(sockaddr_in_t addr);

 /**
 * @brief Transmits data over a TCP socket.
 * @param sock The socket descriptor.
 * @param buffer A pointer to the data buffer to send.
 * @param length The number of bytes to send.
 */
void hl_wifi_tcp_tx(int sock, void* buffer, uint16_t length);

#endif // HL_WIFI_H
```

You've provided two code snippets, which appear to be related to Wi-Fi functionality on an ESP32 microcontroller using the ESP-IDF framework. Let's break down each one.

File 1: `hl_wifi.c` (and its implied `hl_wifi.h`) - A Custom Abstraction Layer

This pair of files (`hl_wifi.c` and its corresponding `hl_wifi.h` which you provided in a previous query) creates a **high-level abstraction** for Wi-Fi station (client) mode and basic TCP client operations. It's designed to simplify common networking tasks, making your main application code cleaner and more focused on its specific logic.

Purpose

The primary purpose of `hl_wifi.c` is to:

- **Encapsulate Wi-Fi setup complexity:** It handles the initialization of the network interface, event loop, Wi-Fi driver, and event handlers.
- **Provide automatic reconnection:** If the Wi-Fi connection drops, it automatically tries to reconnect.
- **Offer a callback mechanism:** It uses a function pointer (`connect_callback_t`) to notify your application when the ESP32 successfully obtains an IP address, which is the signal that networking is ready.
- **Simplify TCP client operations:** It provides helper functions to create network addresses, establish TCP connections, and reliably send data over TCP sockets.

Key Components of `hl_wifi.c`

1. **Includes**: It brings in essential ESP-IDF Wi-Fi, event loop, logging, and network interface headers, along with standard C libraries for string manipulation (`string.h`), internet address conversions (`arpa/inet.h`), and error reporting (`errno.h`).
2. **`TAG`**: A logging tag, `hl_wifi`, used to identify log messages originating from this module.
3. **`event_handler(void* arg, esp_event_base_t event_base, int32_t event_id, void* event_data)`**:
    - This is a static (internal) function that registers with the **ESP-IDF event loop**. It acts as a listener for important Wi-Fi and IP-related events.
    - **`WIFI_EVENT_STA_START`**: When the Wi-Fi station mode starts, it automatically calls `esp_wifi_connect()` to initiate a connection to the configured access point.
    - **`WIFI_EVENT_STA_DISCONNECTED`**: If the ESP32 loses its Wi-Fi connection, this handler will automatically call `esp_wifi_connect()` again to attempt a reconnection. It logs a "retry to connect" message.
    - **`IP_EVENT_STA_GOT_IP`**: This is the crucial event indicating success. When the ESP32 receives an IP address from the router, it logs the IP address. More importantly, it retrieves the `connect_callback_t` function pointer (which was passed during `hl_wifi_init`) and executes it. This is how `hl_wifi` tells your application, "Hey, Wi-Fi is connected and ready to go!"
4. **`hl_wifi_init(connect_callback_t callback)`**:
    - This is the main function you'd call from your `app_main` to start the Wi-Fi connection process.
    - It initializes the **ESP-NETIF** component (`esp_netif_init()`) and the **default event loop** (`esp_event_loop_create_default()`), which are fundamental for networking.
    - It creates a default Wi-Fi station interface (`esp_netif_create_default_wifi_sta()`).
    - It initializes the **ESP Wi-Fi driver** with default settings (`esp_wifi_init(&cfg)`).
    - It registers the `event_handler` to listen for Wi-Fi and IP events. Notice that for the `IP_EVENT_STA_GOT_IP` event, it passes the `callback` function pointer received as an argument, so that `event_handler` can later execute it.
    - It configures the Wi-Fi credentials (`ssid` and `password`) using `CONFIG_ESP_WIFI_SSID` and `CONFIG_ESP_WIFI_PASSWORD`. These are typically defined in your project's `sdkconfig` file, allowing you to set them via `idf.py menuconfig`. It also sets the authentication mode and PMF (Protected Management Frames) settings.
    - Finally, it sets the Wi-Fi mode to Station (`WIFI_MODE_STA`), applies the configuration, and starts the Wi-Fi driver (`esp_wifi_start()`).
5. **`hl_wifi_make_addr(char* ip_str, uint16_t port)`**:

- A helper function to easily create a `sockaddr_in_t` (which is an alias for `struct sockaddr_in`) structure. This structure is used to hold an IP address and port number for network communication.
- It uses `inet_addr()` to convert an IP address string (like "192.168.1.100") into a network byte order integer.
- `htons()` converts the port number to network byte order.

6. **`hl_wifi_tcp_connect(sockaddr_in_t addr)`**:
   - Attempts to establish a **TCP client connection** to the address specified in `addr`.
   - It creates a new socket using the `socket()` function.
   - It then tries to `connect()` the created socket to the remote server.
   - Includes error handling and logging, returning the socket descriptor (an integer) on success, or -1 on failure.
7. **`hl_wifi_tcp_tx(int sock, void* buffer, uint16_t length)`**:
   - Sends a specified `length` of `buffer` data over the given TCP `sock`.
   - This function includes a loop and `offset` variable to ensure **all data is sent**, even if the `send()` system call performs a "partial send" (sends only a portion of the requested data at once).
   - Provides error handling and logging for transmission failures.

File 2: Standard ESP-IDF Wi-Fi Station Example (`main.c` or similar)

This code snippet represents a **standard, self-contained example** of how to connect an ESP32 to a Wi-Fi Access Point using the ESP-IDF framework. This is typically found in the `examples/wifi/station` directory of the ESP-IDF.

Purpose

The primary purpose of this example is to:

- Demonstrate the **minimal steps required** to get an ESP32 connected to a Wi-Fi network in station mode.
- Show how to use **FreeRTOS event groups** for synchronization between the Wi-Fi event handler and the main application logic, waiting for a connection or failure.
- Utilize **Kconfig** for easy configuration of Wi-Fi credentials and retry limits via `idf.py menuconfig`.

Key Components

1. **Includes**: It includes necessary ESP-IDF Wi-Fi, event, and logging headers, along with FreeRTOS components (`freertos/FreeRTOS.h`, `freertos/task.h`, `freertos/event_groups.h`) for synchronization.
2. **Configuration Macros**:
   - `EXAMPLE_ESP_WIFI_SSID`, `EXAMPLE_ESP_WIFI_PASS`, `EXAMPLE_ESP_MAXIMUM_RETRY`: These are defined to use values pulled from **Kconfig** (`CONFIG_ESP_WIFI_SSID`, etc.), which is how users configure their Wi-Fi network details during the build process.
   - `ESP_WIFI_SAE_MODE`, `ESP_WIFI_SCAN_AUTH_MODE_THRESHOLD`: These elaborate conditional defines handle different Wi-Fi security protocols (WPA2, WPA3-SAE, etc.) based on Kconfig selections, ensuring the correct authentication mode is used.

3. `s_wifi_event_group`: A **FreeRTOS event group** handle. This is a synchronization primitive used to signal events between tasks or event handlers.
    - `WIFI_CONNECTED_BIT` (`BIT0`): Set when the ESP32 successfully connects and gets an IP.
    - `WIFI_FAIL_BIT` (`BIT1`): Set when the connection fails after the maximum number of retries.
4. `s_retry_num`: A static integer to keep track of connection retry attempts.
5. `event_handler(void* arg, esp_event_base_t event_base, int32_t event_id, void* event_data)`:
    - Similar to `hl_wifi.c`, this function is registered with the ESP-IDF event loop.
    - `WIFI_EVENT_STA_START`: Calls `esp_wifi_connect()` to start the connection.
    - `WIFI_EVENT_STA_DISCONNECTED`: If disconnected, it checks `s_retry_num`. If retries are still available, it calls `esp_wifi_connect()` again and increments `s_retry_num`. If retries are exhausted, it sets `WIFI_FAIL_BIT` in the event group, signaling a connection failure.
    - `IP_EVENT_STA_GOT_IP`: When an IP is obtained, it logs the IP address, resets `s_retry_num` to 0, and importantly, sets `WIFI_CONNECTED_BIT` in the event group.
6. `wifi_init_sta(void)`:
    - This function encapsulates the Wi-Fi initialization process.
    - It creates the `s_wifi_event_group`.
    - Initializes ESP-NETIF, the default event loop, and creates the Wi-Fi station network interface.
    - Initializes the ESP Wi-Fi driver.
    - Registers the `event_handler` for both `WIFI_EVENT` and `IP_EVENT`.
    - Configures the Wi-Fi SSID, password, and authentication mode based on the Kconfig macros.
    - Sets the Wi-Fi mode to Station, applies the configuration, and starts the Wi-Fi driver.
    - **Crucially, it then calls `xEventGroupWaitBits()`**. This function makes the `wifi_init_sta` function (and by extension, `app_main` if it calls `wifi_init_sta` directly) **block** until either `WIFI_CONNECTED_BIT` or `WIFI_FAIL_BIT` is set by the `event_handler`. This ensures `app_main` knows the outcome of the Wi-Fi connection attempt before proceeding.
    - Based on the bits received from the event group, it logs whether the connection was successful or failed.
7. `app_main(void)`:
    - The entry point of the ESP-IDF application.
    - It initializes **NVS Flash** (`nvs_flash_init()`), which is required by the Wi-Fi driver to store persistent configuration data.
    - Logs the Wi-Fi mode and then calls `wifi_init_sta()` to handle the entire Wi-Fi connection process.

Summary of Differences

The core difference is that the **first example is a full, direct implementation** of Wi-Fi station mode using raw ESP-IDF and FreeRTOS APIs, demonstrating the event group synchronization pattern.

The **second set of files (`hl_wifi.c` and `hl_wifi.h`) is a custom library** that **abstracts away** much of that raw complexity. It provides a simpler, callback-based interface for Wi-Fi connection and adds common TCP client functionalities. Your main application would then use these `hl_wifi_` functions rather than directly interacting with the lower-level ESP-IDF Wi-Fi APIs. This promotes code modularity and readability.