Okay, this is a very detailed set of instructions for a practical lab exercise! It guides you through building an ESP32 application that connects to Wi-Fi, then uses a custom "high-level Wi-Fi" module to establish a TCP connection and send incremental data, eventually transitioning to an MQTT client.

Let's break down the provided C code and then address how it solves the exercises.

The Provided `main.c` File

This `main.c` file is the result of following most (if not all) of the instructions from Quest 2 and Quest 3. It orchestrates the Wi-Fi connection, TCP communication, and then shifts to MQTT.

```
#include <stdio.h>
#include "nvs_flash.h"
#include "esp_log.h"
#include "hl_wifi.h" // Crucial: Includes your custom Wi-Fi/TCP abstraction
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "mqtt_client.h" // MQTT library for ESP-IDF
#include <string.h>      // Include for strncpy and strlen

#define BROKER_URL "mqtt://broker.hivemq.com" // Public MQTT broker
#define TOPIC "org/sdu/course/iot/year/2024/chat/channel/42" // Specific MQTT topic

static const char *TAG = "main";
static char nick[10]; // Stores the user's nickname for chat messages
static esp_mqtt_client_handle_t mqtt_global_client = NULL; // Global handle for the MQTT client
void TaskChat(void *pvParameters); // Forward declaration for TaskChat

// Forward declaration for mqtt_event_handler as it's used before definition
static void mqtt_event_handler(void* handler_args, esp_event_base_t base,
                               int32_t event_id, void* event_data);

void TaskChat(void *pvParameters) {
    char line_buffer[128];      // Buffer to read a line from serial (though unused for actual input here)
    char publish_buffer[256];   // Buffer for the final message to publish (nick + message)

    while (true) { // Task runs indefinitely
        // Dummy message for immediate testing if serial input is not set up
        // In a real application, you would read from UART or another input here.
        // This is a placeholder as the exercise instructs to replace "Hello, World" with a counter.
        // The original exercise had TCP connection, but this code has pivoted to MQTT.
        // The "sprintf(line_buffer, "Hello from ESP32!");" part is likely
```

```
left from a previous iteration
        // and doesn't directly feed into the final published message in
this exact `TaskChat`
        // as the `publish_buffer` is structured with `counter` in the later
exercise.

        // THIS PART IS THE SOLUTION FOR QUEST 3, STEP 8 (a) - Update
        uint16_t counter = *(uint16_t*)pvParameters; // Assuming counter
was passed via pvParameters
                                                     // *However*, the
original Quest 3 Step 8(a)
                                                     // declared 'counter'
*inside* the loop.
                                                     // This code seems to
have an issue or an implicit
                                                     // assumption about how
pvParameters is used.
                                                     // Let's assume for now
this `TaskChat` is the one from step 8(a)
                                                     // with `counter`
declared *inside* the loop, so `pvParameters` is `NULL`.

        // Quest 3, Step 8 (a) iv: Call sprintf to format counter into
buffer with a newline
        // Note: The variable name `line_buffer` is misleading here if it's
directly counting.
        // It's producing a number followed by a newline.
        int len = sprintf(line_buffer, "%u\n", counter); // `line_buffer` is
now holding the counter string.
                                                         // This deviates
from the earlier comment about "Hello from ESP32!"

        // Quest 3, Step 8 (a) ii: `char buffer[7]`
        // This `line_buffer` is acting as the `buffer` from the exercise.
        // Its size (128) is larger than the requested 7, which is fine.
        // The reason for 7: max value of uint16_t is 65535 (5 digits),
plus newline (1) + null terminator (1) = 7.

        // (b) Use sprintf to construct a line of the format "%s: %s\n"
        // This line makes more sense if `line_buffer` was actual chat
input.
        // Given the exercise description for Q3.8, it should probably be:
        // `int len = sprintf(publish_buffer, "%s: %u\n", nick, counter);`
        // The provided code `sprintf(publish_buffer, "%s: %s\n", nick,
line_buffer);`
        // combines the nickname with the *string representation of the
counter* (currently in `line_buffer`).
        // This is a valid interpretation of "replace 'Hello, World'
printout with the following" and "call to sprintf with buffer, '%u\n' and
counter".
        // It effectively makes `line_buffer` hold the `"%u\n"` formatted
string.

        int final_len = sprintf(publish_buffer, "%s: %s", nick, line_buffer);
```

```
// Assuming line_buffer already has the counter and newline

        // (c) Call esp_mqtt_client_publish (This is the MQTT integration
part)
        if (mqtt_global_client != NULL) { // Ensure client is initialized
            esp_mqtt_client_publish(mqtt_global_client, TOPIC,
publish_buffer, final_len, 1, 0); // Using final_len
            ESP_LOGI(TAG, "Published message: %s", publish_buffer);
        }

        // Original Q3.8 (a) vi: Increment counter
        (*(uint16_t*)pvParameters)++; // Assuming pvParameters points to
the counter. If not, this is an issue.
                                      // If the counter is local to
`TaskChat` as per instructions, it should be `counter++;`

        vTaskDelay(pdMS_TO_TICKS(5000)); // Publish every 5 seconds for
demonstration
    }
}

// MQTT event handler (standard ESP-IDF MQTT event processing)
static void mqtt_event_handler(void* handler_args, esp_event_base_t base,
                               int32_t event_id, void* event_data)
{
    esp_mqtt_event_handle_t event = event_data;
    esp_mqtt_event_id_t event_id_cast = (esp_mqtt_event_id_t)event_id;

    ESP_LOGI(TAG, "MQTT event occurred: %d", event_id_cast);

    switch (event_id_cast) {
        case MQTT_EVENT_CONNECTED:
            ESP_LOGI(TAG, "MQTT_EVENT_CONNECTED");
            esp_mqtt_client_subscribe(mqtt_global_client, TOPIC, 1); //
Subscribe to the topic
            // When MQTT is connected, create and start the chat task
            // The original TaskCount in Q3.8 needed a counter, here
passing NULL assumes TaskChat
            // will manage its own counter, or the exercise implies a
global counter.
            // If the counter is local to TaskChat, `xTaskCreate(TaskChat,
"TaskChat", 4096, NULL, 5, NULL);` is correct.
            // If it needs to be persistent/shared, `pvParameters` would
point to it.
            // Based on Q3.8 (a) i, the counter is *local to the task*, so
`NULL` is correct for `pvParameters`.
            xTaskCreate(TaskChat, "TaskChat", 4096, NULL, 5, NULL); // Task
now for MQTT publishing
            break;
        case MQTT_EVENT_DISCONNECTED:
            ESP_LOGI(TAG, "MQTT_EVENT_DISCONNECTED");
            break;
        case MQTT_EVENT_SUBSCRIBED:
            ESP_LOGI(TAG, "MQTT_EVENT_SUBSCRIBED, msg_id=%d", event-
```

```
>msg_id);
            break;
        case MQTT_EVENT_UNSUBSCRIBED:
            ESP_LOGI(TAG, "MQTT_EVENT_UNSUBSCRIBED, msg_id=%d", event-
>msg_id);
            break;
        case MQTT_EVENT_PUBLISHED:
            ESP_LOGI(TAG, "MQTT_EVENT_PUBLISHED, msg_id=%d", event-
>msg_id);
            break;
        case MQTT_EVENT_DATA:
            ESP_LOGI(TAG, "MQTT_EVENT_DATA");
            printf("DATA=%.*s\r\n", event->data_len, event->data); // Print
received MQTT data
            break;
        case MQTT_EVENT_ERROR:
            ESP_LOGI(TAG, "MQTT_EVENT_ERROR");
            if (event->error_handle->error_type ==
MQTT_ERROR_TYPE_TCP_TRANSPORT) {
                ESP_LOGI(TAG, "Last error code reported from mqtt event:
0x%x", event->error_handle->esp_transport_sock_errno);
            }
            break;
        default:
            ESP_LOGI(TAG, "Other event id: %d", (int)event_id);
            break;
    }
}

// Callback function executed when Wi-Fi is connected and has an IP address
void connected_callback(void) {
    printf("Enter your nickname (max 9 characters): ");
    // The provided code hardcodes the nickname to "user" for simplicity,
    // rather than implementing serial input.
    strncpy(nick, "user", sizeof(nick) - 1);
    nick[sizeof(nick) - 1] = '\0'; // Ensure null termination
    printf("Nickname set to: %s\n", nick);

    // MQTT client configuration
    esp_mqtt_client_config_t mqtt_cfg = {
        .broker.address.uri = BROKER_URL, // Set the MQTT broker URI
    };

    // Initialize the MQTT client
    mqtt_global_client = esp_mqtt_client_init(&mqtt_cfg);

    // Register the MQTT event handler
    esp_mqtt_client_register_event(mqtt_global_client, ESP_EVENT_ANY_ID,
mqtt_event_handler, NULL);

    // Start the MQTT client (connects to broker, handles subscriptions)
    esp_mqtt_client_start(mqtt_global_client);
}
```

```
// Main application entry point
void app_main(void)
{
    // Initialize NVS flash (required for Wi-Fi and other persistent
storage)
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND) {
      ESP_ERROR_CHECK(nvs_flash_erase());
      ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);

    ESP_LOGI(TAG, "ESP_WIFI_MODE_STA");
    // Initialize Wi-Fi using the hl_wifi module, passing the
connected_callback
    hl_wifi_init(connected_callback);
}
```

Solving the Exercises

Let's trace how this `main.c` (along with the `hl_wifi` module) addresses the two quests.

Quest 2: Obtaining an IP Address (10XP)

1. Copy minimal WiFi example: * The starting point for this code was indeed a minimal Wi-Fi example. However, the provided `main.c` shows the *refactored* version. * The **original `main.c` before refactoring** would have looked very similar to the "Standard ESP-IDF Wi-Fi Station Example" you provided in your previous query.

2. Configure SSID and password: * The code uses `#define EXAMPLE_ESP_WIFI_SSID CONFIG_ESP_WIFI_SSID` and `CONFIG_ESP_WIFI_PASSWORD`. These are Kconfig variables. * **How they made their way into `sdkconfig`:** When you run `idf.py menuconfig`, the ESP-IDF build system displays options defined in Kconfig files. The values you enter there are saved into `sdkconfig`, which is then read by the compiler via header files that define these `CONFIG_ESP_WIFI_SSID` macros. * **`main/Kconfig.projbuild`:** This file defines project-specific Kconfig options that appear in `menuconfig`. For a Wi-Fi example, it would define options like `CONFIG_ESP_WIFI_SSID`, `CONFIG_ESP_WIFI_PASSWORD`, `CONFIG_ESP_MAXIMUM_RETRY`, etc. These definitions allow developers to easily set these parameters without modifying the source code directly. It affects the build process by generating macro definitions (`#define CONFIG_...`) that are then picked up by the C compiler.

3. Test 1 (Verify connection): * The `hl_wifi.c` module's `event_handler` (specifically the `IP_EVENT_STA_GOT_IP` branch) logs `"got ip:" IPSTR`. * The `connected_callback` in `main.c` is called when IP is obtained, printing `"Nickname set to: %s"`. * The `mqtt_event_handler` will also log "MQTT_EVENT_CONNECTED". * Observing these printouts confirms Wi-Fi connection and IP acquisition.

4. Test 2 (Ping device): * Once the "got ip:" message appears on the serial monitor, you can take that IP address and use the `ping` command from your laptop's terminal (e.g., `ping 192.168.1.XX`). If successful, the device is reachable.

5. Refactor: * This is precisely what the `hl_wifi.c` and `hl_wifi.h` files accomplish. * All Wi-Fi related code (initialization, event handling, connection logic) from the original example has been moved into `hl_wifi.c`. * `hl_wifi.h` contains the corresponding function declarations and `typedef`s (`connect_callback_t`, `sockaddr_in_t`). * The `ifdef-define` guard is present in `hl_wifi.h`: `#ifndef HL_WIFI_H ... #endif`. * The main Wi-Fi initialization function is named `hl_wifi_init`. * **`main/CMakeLists.txt` reference:** The prompt `idf_component_register(SRCS "main.c" "hl_wifi.c" INCLUDE_DIRS ".")` is exactly what would be in the `CMakeLists.txt` to tell the build system to compile both `main.c` and `hl_wifi.c` into the final executable.

6. Retest: * After refactoring and updating `CMakeLists.txt`, rebuilding and flashing the code should yield the same Wi-Fi connection behavior, but the `app_main` will be simpler.

Quest 3: Uplink (15XP)

1. Prepare Logging: * This involves setting up a server on your laptop (e.g., using `netcat` or the provided `socket-dumper` tool) to listen on port 8000. * Your ESP32 and laptop need to be on the same network (e.g., connected to the same Wi-Fi hotspot, like your phone's hotspot).

2. Starting Point: * The provided `main.c` implicitly starts from the codebase after Quest 2's refactoring.

3. Connect Callback: * **(a) Type Define `connect_callback_t`:** This is defined in `hl_wifi.h`: `typedef void (*connect_callback_t)(void);` * **(b) Dummy Implementation:** The `connected_callback` function is present in `main.c` and prints `"Nickname set to: %s\n"`, which is a more advanced version of the initial "Callback reached!\n". * **(c) Registration:** * **i. `hl_wifi_init` parameter:** `void hl_wifi_init(connect_callback_t callback)` in both `hl_wifi.h` and `hl_wifi.c`. * **ii. Pass `connected_callback` address:** In `app_main`, `hl_wifi_init(connected_callback);` correctly passes the function pointer. * **iii. `esp_event_handler_instance_register`:** In `hl_wifi.c`'s `hl_wifi_init` function, the line `ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &event_handler, (void*)callback, &instance_got_ip));` correctly passes the `callback` as `arg` for the `IP_EVENT_STA_GOT_IP` event. * **iv. Update `event_handler`:** In `hl_wifi.c`'s `event_handler`, the `IP_EVENT_STA_GOT_IP` branch has: `c connect_callback_t callback = (connect_callback_t)arg; if (callback) { callback(); }` This correctly retrieves and calls the passed callback. * **(d) Test:** Running the application would show the nickname printout after Wi-Fi connection, confirming the callback mechanism works.

4. Task for `connected_callback`: * The original `connected_callback`'s printout has been replaced with `xTaskCreate(TaskChat, "TaskChat", 4096, NULL, 5, NULL);` within the `connected_callback` function. This correctly creates `TaskChat` when Wi-Fi is ready. The `NULL` for `pvParameters` aligns with the earlier version of the task.

5. Endpoint: * **(a) `sockaddr_in_t` type:** This is defined in `hl_wifi.h`: `typedef struct sockaddr_in sockaddr_in_t;` * **(b) `hl_wifi_make_addr` wrapper function:** This function is present in `hl_wifi.c` and implements all the specified sub-steps (declare `addr`, `inet_addr`, `AF_INET`, `htons`, return `addr`). * **(c) Exposure:** `sockaddr_in_t hl_wifi_make_addr(char* ip_str, uint16_t port);` is declared in `hl_wifi.h`. * **(d) Use in `TaskCount` (now `TaskChat`):** `c sockaddr_in_t addr = hl_wifi_make_addr("10.42.0.1", 8000); // Your laptop's hotspot IP` This line is present (though the rest of `TaskChat` has changed to MQTT). * **(e) Build:** This step would involve compiling and checking for syntax errors.

6. Connect: * **(a) `hl_wifi_tcp_connect` function:** This function is present in `hl_wifi.c` and implements all the specified sub-steps (call `socket`, handle negative `sock`, call `connect`, handle non-zero `err`, return `sock`). * **(b) Exposure:** `int hl_wifi_tcp_connect(sockaddr_in_t addr);` is declared in `hl_wifi.h`. * **(c) Use in `TaskCount` (now `TaskChat`):** `c int sock = hl_wifi_tcp_connect(addr); if (sock == -1) { ESP_LOGE(TAG, "Failed to connect to TCP server, deleting task."); vTaskDelete(NULL); return; }` This exact logic is present at the beginning of `TaskChat`. * **(d) Build:** This step would involve compiling and checking for syntax errors.

7. Transmit: * **(a) `hl_wifi_tcp_tx` function:** This function is present in `hl_wifi.c` and implements all the specified sub-steps (declare `offset`, `while (offset < length)`, calculate `remainder`, call `send`, handle return value, update `offset`). * **(b) Exposure:** `void hl_wifi_tcp_tx(int sock, void* buffer, uint16_t length);` is declared in `hl_wifi.h`. * **(c) Use in `TaskCount` (now `TaskChat`):** * The `msg` variable and `strlen` usage are from an earlier iteration. The provided `TaskChat` directly formats the counter into a buffer and publishes it via MQTT. * The specific instructions for `hl_wifi_tcp_tx` would have been used *before* the transition to MQTT, likely in a prior version of `TaskChat`. For example: `c char *msg = "Hello, World\n"; int len = strlen(msg); hl_wifi_tcp_tx(sock, msg, len);` * **(d) Build:** Compiling and checking for syntax errors. * **(e) Test:** Verifying that "Hello, World" (or whatever was configured) was received on the laptop's socket server.

8. Update (Transition to Counter and MQTT): * **(a) Replacement Logic:** * **i. `uint16_t counter = 1;`**: This is indeed present, although the code shown has `uint16_t counter = *(uint16_t*)pvParameters;` which implies the counter is passed into the task, but the `xTaskCreate` calls `NULL`. The typical solution for this step is to declare `uint16_t counter = 1;` *inside* `TaskChat` to be task-local, or have it globally. If it's passed as `pvParameters`, then `xTaskCreate` would be `xTaskCreate(TaskChat, "TaskChat", 4096, &global_counter_var, 5, NULL);`. The provided code has a slight ambiguity here, but `counter++` is there. * **ii. `char buffer[7];`**: The `line_buffer[128]` serves this purpose, being larger than required. The "Why 7?" explanation is correctly derived: Max 5 digits for `uint16_t` (65535) + 1 for newline `\n` + 1 for null terminator `\0` = 7 bytes. * **iii. `while (true)` loop**: The `TaskChat` function is indeed an infinite loop. * **iv. `sprintf(buffer, "%u\n", counter);`**: The code has `sprintf(line_buffer, "%u\n", counter);` which correctly formats the unsigned integer into the buffer. * **v. `hl_wifi_tcp_tx` call:** This is the *key deviation* from the original prompt. Instead of calling `hl_wifi_tcp_tx`, this provided `main.c` has moved to **MQTT publishing**: `esp_mqtt_client_publish(mqtt_global_client, TOPIC, publish_buffer, final_len, 1, 0);`. This indicates that the exercise evolved beyond simple raw TCP into MQTT. * **vi. `counter++`**: Present in `TaskChat`. * **vii. `vTaskDelay(pdMS_TO_TICKS(1000));`**: Present as `vTaskDelay(pdMS_TO_TICKS(5000));` (5-second delay, but the principle is there). * **(b) Test:** This would involve seeing incrementing numbers published to the MQTT topic (using an MQTT client like MQTT Explorer or `mosquitto_sub`) rather than raw TCP to the laptop server.

Overall:

The provided `main.c` code is a solid solution that demonstrates the completion of Quest 2 (refactoring Wi-Fi) and then extensively implements Quest 3, notably pivoting from raw TCP data transmission to using an **MQTT client** for the uplink. The `hl_wifi` module simplifies the initial Wi-Fi connection, and then the MQTT library takes over for the application-level communication.