HCMC University Of Technology, VNU-HCM
Faculty of Computer Science & Engineering

– Embedded System Lab 06 –

ESP32: WiFi Subsystem

Pham Hoang Anh & **Huynh Hoang Kha**

Goal In this lab, students are expected to understand and be able to configure ESP32 WiFi subsystem as:

- An Access Point
- A Station

Content

- Initializing and setting the operation mode
- WiFi operations

Prerequisite Requirement

- Have basic knowledge of computer networking

Grading policy

- 40% in-class performance
- 60% report submission

1 Initializing and setting the operation mode

An ESP32 device can play the role of an Access Point, a Station or both at the same time.

1.1 Initializing the WiFi environment

WiFi is only a part of the capabilities of an ESP32. As such, there may be some times when you actually don't want to use the WiFi subsystem. To accommodate those patterns, the initialization of the WiFi subsystem is expected to be performed by you when you write your applications. This is done by calling the `esp_wifi_init()` method. The recommended way of doing this is as follows:

```
1 wifi_init_config_t config = WIFI_INIT_CONFIG_DEFAULT();
2 esp_wifi_init(&config);
```

1.2 Setting the operation mode

The ESP32 can either be a station in the network, an access point for other devices or both. Remember, when an ESP32 is being a station, it can connect to a remote access point (your WiFi hub) while when being an access point, other WiFi stations can connect to the ESP32 (think of the ESP32 as becoming a WiFi hub). This is a fundamental consideration and we will want to choose how the device behaves early on in our application design. Once we have chosen what we want, we set a global mode property which indicates which of the operational modes our device will perform (station, access point or station AND access point).

This choice is set with a call to `esp_wifi_set_mode()`. The parameter is an instance of `wifi_mode_t` which can have a value of `WIFI_MODE_NULL`, `WIFI_MODE_STA`, `WIFI_MODE_AP` or `WIFI_MODE_APSTA`. We can call `esp_wifi_get_mode()` to retrieve our current mode state.

2 WiFi Operations

2.1 Handling WiFi Events

During the course of operating as a WiFi device, certain events may occur that ESP32 needs to know about. These may be of importance or interest to the applications running within it. Since we don't know when, or even if, any events will happen, we can't have our application block waiting for them to occur. Instead what we should do is define a callback function that will be invoked should an event actually occur. The function called `esp_event_loop_init()` does just that. It registers a function that will be called when the ESP32 detects certain types of WiFi related events. The registered function is invoked and passed a rich

data structure that includes the type of event and associated data corresponding to that event. The types of events that cause the callback to occur are:

- We connected to an access point
- We disconnected from an access point
- The authorization mode changed
- A station connected to us when we are in Access Point mode
- A station disconnected from us when we are in Access Point mode
- A SSID scan completes

When the ESP32 WiFi environment operates, it publishes "events" when something at the WiFi level occurs such as a new station connecting. We can register a callback function that is invoked when an event is published. The signature of the callback function is:

```
esp_err_t eventHandler(void* ctx, system_event_t* event) {
   // Handle event here ...
   return ESP_OK;
}
```

Typically, we need to also include the following:

```
#include <esp_event.h>
#include <esp_event_loop.h>
#include <esp_wifi.h>
#include <esp_err.h>
```

To register the callback function, we invoke: `esp_event_loop_init(eventHandler, NULL);`. If we wish to subsequently change the event handler associated with our WiFi handling we can call: `esp_event_loop_set_cb(eventHandler, NULL);`

When the event handler is invoked, the event parameter is populated with details of the event. The data type of this parameter is a `system_event_t`n which contains:

```
system_event_id_t       event_id
system_event_info_t     event_info
```

Detailed references to `system_event_t` could be found at:

https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/system/esp_event_legacy.html#_CPPv414system_event_t

2.2 Scanning for access points

If the ESP32 is going to be performing the role of a station we will need to connect to an access point. We can request a list of the available access points

against which we can attempt to connect. We do this using the `esp_wifi_scan_start()` function.

The results of a WiFi scan are stored internally in ESP32 dynamically allocated storage. The data is returned to us when we call `esp_wifi_scan_get_ap_records()` which also releases the internally allocated storage. As such, this should be considered a destructive read. A scan record is contained in an instance of a `wifi_ap_record_t` structure that contains:

```
uint8_t                 bssid[6]
uint8_t                 ssid[32]
uint8_t                 primary
wifi_second_chan_t      second
int8_t                  rssi
wifi_auth_mode_t        authmode
```

The `wifi_auth_mode_t` can be one of the followings pre-defined constants: `WIFI_AUTH_OPEN` (No security), `WIFI_AUTH_WEP` (WEP security), `WIFI_AUTH_WPA_PSK` (WPA security), `WIFI_AUTH_WPA2_PSK` (WP2 security), and `WIFI_AUTH_WPA_WPA2_PSK` (WPA or WPA2 security), respectively.

After issuing the request to start performing a scan, we will be informed that the scan completed when a `SYSTEM_EVENT_SCAN_DONE` event is published. The event data contains the number of access points found but that can also be retrieved with a call to `esp_wifi_scan_get_ap_num()`. Should we wish to cancel the scanning before it completes on its own, we can call `esp_wifi_scan_stop()`.

```
#include "esp_wifi.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "nvs_flash.h"

esp_err_t event_handler(void *ctx, system_event_t *event)
{
   if (event->event_id == SYSTEM_EVENT_SCAN_DONE) {
      printf("Number of access points found: %d\n",
         event->event_info.scan_done.number);
      uint16_t apCount = event->event_info.scan_done.number;
      if (apCount == 0) {
         return ESP_OK;
      }
      wifi_ap_record_t *list =
        (wifi_ap_record_t *)malloc(sizeof(wifi_ap_record_t) * apCount);
      ESP_ERROR_CHECK(esp_wifi_scan_get_ap_records(&apCount, list));
      int i;
      for (i=0; i<apCount; i++) {
         char *authmode;
         switch(list[i].authmode) {
            case WIFI_AUTH_OPEN:
               authmode = "WIFI_AUTH_OPEN";
               break;
            case WIFI_AUTH_WEP:
               authmode = "WIFI_AUTH_WEP";
               break;
```

```
            case WIFI_AUTH_WPA_PSK:
               authmode = "WIFI_AUTH_WPA_PSK";
               break;
            case WIFI_AUTH_WPA2_PSK:
               authmode = "WIFI_AUTH_WPA2_PSK";
               break;
            case WIFI_AUTH_WPA_WPA2_PSK:
               authmode = "WIFI_AUTH_WPA_WPA2_PSK";
               break;
            default:
               authmode = "Unknown";
               break;
         }
         printf("ssid=%s, rssi=%d, authmode=%s\n",
            list[i].ssid, list[i].rssi, authmode);
      }
      free(list);
   }
   return ESP_OK;
}

int app_main(void)
{
   nvs_flash_init();
   tcpip_adapter_init();
   ESP_ERROR_CHECK(esp_event_loop_init(event_handler, NULL));
   wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
   ESP_ERROR_CHECK(esp_wifi_init(&cfg));
   ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
   ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
   ESP_ERROR_CHECK(esp_wifi_start());
   // Let us test a WiFi scan ...
   wifi_scan_config_t scanConf = {
      .ssid = NULL,
      .bssid = NULL,
      .channel = 0,
      .show_hidden = 1
   };
   ESP_ERROR_CHECK(esp_wifi_scan_start(&scanConf, 0));
   return 0;
}
```

Code 1: A WiFi scan example

2.3 Starting up the WiFi environment

Since WiFi has states that it must go through, a question that may be asked is "When is WiFi ready to be used?". If we imagine that an ESP32 boots from cold, the chances are that we want to tell it to be either a station or an access point and then configure it with parameters such as which access point to connect to (if it is a station) or what its own access point identity should be (if it is going to be an access point). Given that these are a sequence of steps, we actually don't want the ESP32 to execute on these tasks until after we have performed all our setup. For example, if we boot an ESP32 and ask it to be

an access point, if it started being an access point immediately then it may not yet know the details of the access point it should be or, worse, may transiently appear as the wrong access point. As such, there is final command that we must learn which is the instruction to the WiFi subsystem to start working. That command is `esp_wifi_start()`. Prior to calling that, all we are doing is setting up the environment. Only by calling `esp_wifi_start()` does the WiFi subsystem start doing any real work on our behalf. If our mode is that of an access point, calling this function will start us being an access point. If our mode is that of a station, now we are allowed to subsequently connect as a station. There is a corresponding command called `esp_wifi_stop()` which stops the WiFi subsystem.

We should not perform any additional networking or WiFi work until we receive the corresponding event (`SYSTEM_EVENT_STA_START` or `SYSTEM_EVENT_AP_START`) to indicate that WiFi has started.

2.4 Connecting to an access point

Once the ESP32 has been set up with the station configuration details, which includes the SSID and password, we are ready to perform a connection to the target access point. The function `esp_wifi_connect()` will form the connection. Realize that this is not instantaneous and you should not assume that immediately following this command you are connected. Nothing in the ESP32 blocks and as such neither does the call to this function. Some time later, we will actually be connected. We will see two callback events fired. The first is `SYSTEM_EVENT_STA_CONNECTED` indicating that we have connected to the access point. The second event is `SYSTEM_EVENT_STA_GOT_IP` which indicates that we have been assigned an IP address by the DHCP server. Only at that point can we truly participate in communications. If we are using static IP addresses for our device, then we will only see the connected event.

Should we disconnect from an access point, we will see a `SYSTEM_EVENT_STA_DISCONNECTED` event. To disconnect from a previously connected access point we issue the `esp_wifi_disconnect()` call.

There is one further consideration associated with connecting to access points and that is the idea of automatic connection. There is a boolean flag that is stored in flash that indicates whether or not the ESP32 should attempt to automatically connect to the last used access point. If set to true, then after the device is started and without you having to code any API calls, it will attempt to connect to the last used access point. This is a convenience that I prefer to switch off. Usually, I want control in my device to determine when I connect. We can enable or disable the auto connect feature by making a call to `esp_wifi_set_auto_connect()`.

```
1 #include <freertos/FreeRTOS.h>
2 #include <esp_wifi.h>
3 #include <esp_system.h>
```

```
#include <esp_event.h>
#include <esp_event_loop.h>
#include <nvs_flash.h>
#include <tcpip_adapter.h>

esp_err_t event_handler(void *ctx, system_event_t *event)
{
    if (event->event_id == SYSTEM_EVENT_STA_GOT_IP)
    {
        printf("Our IP address is " IPSTR "\n",
               IP2STR(&event->event_info.got_ip.ip_info.ip));
        printf("We have now connected to a station and can do things...\n");
    }
    if (event->event_id == SYSTEM_EVENT_STA_START)
    {
        ESP_ERROR_CHECK(esp_wifi_connect());
    }
    return ESP_OK;
}

int app_main(void)
{
    nvs_flash_init();
    tcpip_adapter_init();
    ESP_ERROR_CHECK(esp_event_loop_init(event_handler, NULL));
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));
    ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    wifi_config_t sta_config = {
        .sta = {
            .ssid = "RASPI3",
            .password = "password",
            .bssid_set = 0}};
    ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &sta_config));
    ESP_ERROR_CHECK(esp_wifi_start());
    return 0;
}
```

When we connect to an access point, our device is being a station. The connection to the access point doesn't automatically mean that we now have an IP address. We still have to request an allocated IP address from the DHCP server. This can take a few seconds. In some cases, we can get away with the device requesting a specific IP address. This results in a much faster connection time. If we do specify data, we also need to supply DNS information should we need to connect to DNS servers for name resolution.

```
#include <lwip/sockets.h>
// The IP address that we want our device to have.
#define DEVICE_IP          "192.168.1.99"
// The Gateway address where we wish to send packets.
// This will commonly be our access point.
#define DEVICE_GW          "192.168.1.1"
// The netmask specification.
#define DEVICE_NETMASK     "255.255.255.0"
// The identity of the access point to which we wish to connect.
#define AP_TARGET_SSID     "RASPI3"
```

```
11
12 // The password we need to supply to the access point for authorization.
13 #define AP_TARGET_PASSWORD "password"
14 esp_err_t wifiEventHandler(void *ctx, system_event_t *event)
15 {
16    if (event->event_id == SYSTEM_EVENT_STA_START) {
17       ESP_ERROR_CHECK(esp_wifi_connect());
18    }
19    return ESP_OK;
20 }
21
22 // Code fragment here ...
23   nvs_flash_init();
24   tcpip_adapter_init();
25   tcpip_adapter_dhcpc_stop(TCPIP_ADAPTER_IF_STA); // Don't run a DHCP client
26   tcpip_adapter_ip_info_t ipInfo;
27   inet_pton(AF_INET, DEVICE_IP, &ipInfo.ip);
28   inet_pton(AF_INET, DEVICE_GW, &ipInfo.gw);
29   inet_pton(AF_INET, DEVICE_NETMASK, &ipInfo.netmask);
30   tcpip_adapter_set_ip_info(TCPIP_ADAPTER_IF_STA, &ipInfo);
31   ESP_ERROR_CHECK(esp_event_loop_init(wifiEventHandler, NULL));
32   wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
33   ESP_ERROR_CHECK(esp_wifi_init(&cfg) );
34   ESP_ERROR_CHECK(esp_wifi_set_storage(WIFI_STORAGE_RAM));
35   ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
36   wifi_config_t sta_config = {
37     .sta = {
38       .ssid      = AP_TARGET_SSID,
39       .password  = AP_TARGET_PASSWORD,
40       .bssid_set = 0
41     }
42   };
43   ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &sta_config));
44   ESP_ERROR_CHECK(esp_wifi_start());
```

2.5 Being an access point

So far we have only considered the ESP32 as a WiFi station to an existing access point but it also has the ability to be an access point to other WiFi devices (stations) including other ESP32s.

In order to be an access point, we need to define the SSID that allows other devices to distinguish our network. This SSID can be flagged as hidden if we don't wish it to be found in a scan. In addition, we will also have to supply the authentication mode that will be used when a station wishes to connects with us. This is used to allow authorized stations only and prevent un-authorized ones. Only stations that know our password will be allowed to connect. If we are using authentication, then we will also have to choose a password which the connecting stations will have to know and supply to successfully connect.

The first task in being an access point is to flag the ESP32 as being such using the `esp_wifi_set_mode()` function and pass in the flag that requests we be either a dedicated access point or an access point and a station. This will be either

`esp_wifi_set_mode(WIFI_MODE_AP);` or `esp_wifi_set_mode(WIFI_MODE_APSTA)`.

Next we need to supply the configuration information. We do this by populating an instance of `wifi_ap_config_t`. References: https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/network/esp_wifi.html#_CPPv416wifi_ap_config_t

```
wifi_config_t apConfig = {
   .ap = {
      .ssid="<access point name>",
      .ssid_len=0,
      .password="<password>",
      .channel=0,
      .authmode=WIFI_AUTH_OPEN,
      .ssid_hidden=0,
      .max_connection=4,
      .beacon_interval=100
   }
};
```

With the structure populated, we call `esp_wifi_set_config()` ... for example: `esp_wifi_set_config(WIFI_IF_AP, &apConfig);`. Finally, we call `esp_wifi_start()`.

When we become an access point, an ESP32 WiFi event is produced of type `SYSTEM_EVENT_AP_START`. Note that there is no payload data associated with this event.

Once the ESP32 starts listening for station connects by being an access point, we are going to want to validate that this works. You can use any device or system to scan and connect.

When a station connects, the ESP32 will raise the `SYSTEM_EVENT_AP_STACONNECTED` event. When a station disconnects, we will see the `SYSTEM_EVENT_AP_DISCONNECTED` event.

When an ESP32 acts as an access point, this allows other devices to connect to it and form a WiFi connection. However, it appears that two devices connected to the same ESP32 acting as an access point can not directly communicate between each other. For example, imagine two devices connecting to an ESP32 as an access point. They may be allocated the IP addresses 192.168.4.2 and 192.168.4.3. We might imagine that 192.168.4.2 could ping 192.168.4.3 and visa versa but that is not allowed. It appears that they only direct network connection permitted is between the newly connected stations and the access point (the ESP32) itself. This seems to limit the applicability of the ESP32 as an access point. The primary intent of the ESP32 as an access point is to allow mobile devices (e.g., your phone) to connect to the ESP32 and have a conversation with an application that runs upon it.