Software Technology of Internet of Things

Radio Exercises: That Old Wireless Magic

Aslak Johansen <asjo@mmmi.sdu.dk>

Mar 14, 2022

Quest 1: Establish an Access Point (5XP)

For todays quests you will need a 2.4GHz access point under your control. Modern phones and laptops can all establish such an access point. Pick one, and make sure that whatever connects to this access point can access your laptop. You will need an SSID to identify the network and a password to control access.

Quest 2: Obtaining an IP Address (10XP)

First steps are often the hardest. Getting the initial codebase running, making that first connect, and being handed your very own an IP address. It's a coming of age story, and now it is your time to undertake that journey.

1. ***Copy*** Make a copy of the minimal WiFi example from the esp-idf repository[1].
2. ***Configure*** Follow the instructions in `README.md` to configure SSID and password for your network.
   - How did they make their way into the `sdkconfig` file?
   - Open up the `main/Kconfig.projbuild` file in a text editor. How does file affect the build process (the "menuconfig" step as well as the `main/station_example_main.c` file)?
3. ***Test 1*** Verify that the application connects to your network by observing the serial printouts.
   Note: Most devices have a reset button. Pressing it will ... well ... reset it. This is convenient if you missed some printouts in during the boot sequence.

[1] `https://github.com/espressif/esp-idf/tree/master/examples/wifi/getting_started/station`

4. ***Test 2*** Verify that you can ping the device from your laptop.

5. ***Refactor*** Refactor the code by pulling out all WiFi relevant code into `hl_wifi.c` and create a matching `hl_wifi.h` header file. Make sure that you use an ifdef-define construct to guard against multiple inclusions[2] and name your main wifi initialization function `hl_wifi_init`. Your main C file should include the header file and your `main/CMakeLists.txt` should reference the new "module":

```
idf_component_register(SRCS "main.c"
                            "hl_wifi.c"
                    INCLUDE_DIRS ".")
```

6. ***Retest*** Make sure that you now have a reusable wifi "module" with as simple an interface as possible.

Quest 3: Uplink (15XP)

With a link established the next step is to open a socket to another machine, and start transferring data. The steps involved are, however, not for the faint of heart. Much dark knowledge is needed. You must abstract away this complexity so that future projects will no longer have to bear this burden!

1. ***Prepare Logging*** You will need a few things for this:
   - A program that can timestamp individual lines received over a socket[3].
   - A laptop running this program and no firewall in front of the port bound by it. For the remainder of this text we will assume it to be port 8000.
   - The IoT device on a network capable of reaching the laptop[4].

2. ***Starting Point*** Start out by creating a copy of the code from Quest 2. The remainder of this quest will focus on extending that codebase.

3. ***Connect Callback***
   (a) ***Type*** Define `connect_callback_t` as the type of a pointer to a function that takes not parameters and returns `void`[5]. Declare this type on `hl_wifi.h`.
   (b) ***Dummy Implementation*** Add to the `main.c` file a function called `connected_callback` that matches `connect_callback_t` and prints out `"Callback reached!\n"` when called.

[2] See C slides on the precompiler step if this is confusing.
[3] `https://github.com/aslakjohansen/socket-dumper`
Binaries for popular platforms are available on itslearning.
[4] This can be accomplished by having a cellphone share its internet connection to both laptop and IoT device.
[5] See C slides on function pointers if this is confusing.

(c) ***Registration*** The last step is to make sure that the callback function can be named within the context of the function of our wifi module that determines that the connection has been established.

i. Add a parameter to `hl_wifi_init` of type `connect_callback_t*` and make sure to call that function using the address of `connected_callback` as parameter. Update the function prototype of `hl_wifi.h` and `hl_wifi.c` to reflect the change.

ii. In the declaration of `hl_wifi_init` pass the function pointer onto `wifi_init_sta`.

iii. In the declaration of `wifi_init_sta`, replacing the `NULL` pointer in the fourth argument to the call to `esp_event_handler_instance_register` with the function pointer function pointer. Note that there are two such calls. The one we care about is the one the handles the `IP_EVENT_STA_GOT_IP` event. This registers it as data to be passed to the local event handler.

iv. The next step is to update that local event handler (aka the `event_handler` function). As you can see, it branches on `event_base` and `event_id`. What we care about is the existing branch that has to do with the event id `IP_EVENT_STA_GOT_IP` (remember last sted?). Add to the end of this branch:

```
connect_callback_t callback = arg;
callback();
```

(d) ***Test*** Run the application to make sure that the callback function is being called whenever the connection has been established.

4. ***Task*** For good measure, lets define a task called `TaskCount`. Move the printout statement from `connected_callback` and make sure that `connected_callback` creates this task. Verify that the code does as you expect.

5. ***Endpoint***

(a) ***Type*** For convenience, start by defining a new type `sockaddr_in_t` in `hl_wifi.h` to be an alias for `struct sockaddr_in`.

(b) ***Wrapper Function*** Create a `hl_wifi_make_addr` function in `hl_wifi.c` that takes a `char*` and a `uint16_t` as parameters and returns a `sockaddr_in_t`. In this function:

i. Declare a variable `addr` of type `struct sockaddr_in`.

ii. Pass the `char*` function parameter through the `inet_addr` function and assign the result to `addr.sin_addr.s_addr`.

iii. Assign `AF_INET` to `addr.sin_family`.

iv. Pass the `uint16_t` function parameter through the `htons` function and assign the result to `addr.sin_port`. This converts from host byte order to network byte order.

v. Return `addr`.

(c) ***Exposure*** Expose that function through the corresponding header file.

(d) ***Use*** In `TaskCount`, create a new variable called `addr` of type `sockaddr_in_t` and initialize it to the result of calling `hl_wifi_make_addr` with the IP address of your laptop and port 8000.

(e) ***Build*** Make sure your code builds.

6. ***Connect***

(a) ***Connect Function*** In `hl_wifi.c`, add a function called `hl_wifi_tcp_connect` that takes a `struct sockaddr_in` called `addr` as parameter and returns an `int`. The function body should:

i. Call `socket` with `AF_INET`, `SOCK_STREAM` and `IPPROTO_IP` as parameters. Store the return value in a variable called `sock` of type `int`.

ii. Handle a negative value of `sock` by printing out a human readable text along with the value of `errno`, and returning `-1`. The variable `errno` contains an id for the last error.

iii. Call `connect` with `sock`, the address of `addr` and `sizeof(struct sockaddr_in6)` as parameters.

iv. Capture the return value as `err` of type `int`. Handle a non-zero value of `err` in the same way you handled the error condition for `socket`

v. Return `sock`.

(b) ***Exposure*** Expose that function through the corresponding header file.

(c) ***Use*** In `TaskCount`, make sure to call this function with `addr` as parameter. Check the return type (-1 means failure). In case of failure, report this over the serial link and call `vTaskDelete` with `NULL` as parameter to remove the current task.
Note: Removing the current task does not return from it.

(d) ***Build*** Make sure your code builds.

7. ***Transmit***

(a) ***Transmit Function*** In `hl_wifi.c`, declare a `void` function called `hl_wifi_tcp_tx` that takes three parameters, namely (i) a `socket` of type `int`, a `buffer` of type `void*` and a `length` of type `uint16_t`. Calling this function should transmit `length` starting from the beginning of `buffer` over the connection named by `socket`. In the body of this function:

i. Declare an `offset` variable of type `uint16_t` that you initialize to zero.

ii. The remainder of the code should fit into a while loop with the condition `offset<length`.

iii. First calculate the `remainder` (type `uint16_t`) as `length-offset`.

iv. Then you try to send this number of bytes by calling `send` with four parameters. These are (i) `sock`, (ii) the sum of `buffer` casted to a `char*` and `offset`, (iii) `remainder`, and (iv) zero.

v. The return value of this function call has two functions. Firstly, any negative error indicates an error. You should react to this with an informative error message. Secondly, it tells how many bytes were actually transmitted. This can be anything up until, and including, what you asked it to send.

vi. Update the value of `offset` to reflect how far you have come, so that next iteration will take care of the rest.

(b) ***Exposure*** Expose that function through the corresponding header file.

(c) ***Use*** In `TaskCount` add a call to the newly defined transmit function.

i. Declare a variable `msg` of type `char*` and initialize it to have the value `"Hello, World"`.

ii. Use the `include` precompiler directive to import the declarations of `string.h`.

iii. Get the length of `msg` by calling the `strlen` function on it.
Note: This function works by counting bytes from the address pointed to by its argument until it reaches a zero value. So, it is really only usable if you use zero-terminated strings.

(d) ***Build*** Make sure your code builds.

(e) ***Test*** Verify that your application does as you expect.

8. ***Update***

(a) ***Replacement Logic*** Replace the `"Hello, World"` printout with the following:

i. Declare a 16-bit unsigned integer variable named `counter` and initialize it to a value of 1.

ii. Declare a `char[7]` variable called `buffer`. Why the 7?

iii. Add a while-true loop with the following code in its body:

iv. A call to `sprintf` with `buffer`, `"%u\n"` and `counter` as parameters. What does this do?

v. A call to `hl_wifi_tcp_tx` with `socket`, `buffer` and the length of `buffer` as parameters.

vi. Increment `counter`.

vii. Add a delay of 1s.

(b) ***Test*** Verify that your application does as you expect.

Quest 4: Temperature Uplink (5XP)

Given how well you performed in the last Quest, it should be a small matter to complete this Quest. You simply have to create a variant that sends periodic temperature readings instead of counting. How hard can it be?

1. ***Starting Point*** Make a copy of your result from Quest 3 and call it `temp2tcp`.

2. ***Update*** Remove the silly bit about counting, and replace it with code that samples of your temperature sensor and converts it to a human readable number.
3. ***Test*** Verify that your application does as you expect.

Quest 5: Chatty Room (15XP)

You have proven yourself a worthy student, but only so much can be accomplished alone! It is time to introduce you to the network of AEGIS (the Association of Excellent and Glorious IoT Scholars). But the final step, you will have to walk alone. Complete the following steps, and you will finally be able to connect to AEGIS.

1. ***Documentation*** Locate the minimal MQTT example from the esp-idf repository[6], and keep it handy in case of trouble.
2. ***Copy*** Make a copy of the codebase from Quest 3 and call it `chat`.
3. ***Update*** Initial changes are:
   (a) Include the `mqtt_client.h` header file.
   (b) Remove all contents of the `connected_callback` function.
   (c) Introduce two precomplier definitions:
   - `BROKER_URL` with a value of `("mqtt://broker.hivemq.com")`
   - `TOPIC` with a value of `("org/sdu/course/iot/year/2022/chat/channel/42")`
4. ***Username*** Inside the `connected_callback` function promt the user for a nickname, and store it in a global variable called `nick`. This variable should have the type `char[10]`. What consequences does this typing have for the nickname?
5. ***MQTT Connect*** The chat service is hosted on a public so-called MQTT broker. To communicate with it we first need to connect to it. We do this inside the `connected_callback` function.
   (a) Create a variable called `mqtt_cfg` of type `esp_mqtt_client_config_t` and initialize its `uri` field to have a value of `BROKER_URL`.
   (b) Pass the address of `mqtt_cfg` to `esp_mqtt_client_init` and store the return value of this call in a `client` variable of type `esp_mqtt_client_handle_t`.
   (c) Create an the following function:

```
static void mqtt_event_handler (void*             handler_args,
                                esp_event_base_t  base,
                                int32_t           event_id,
                                void*             event_data)
{
  printf("Receiving hits!\n");
}
```

[6] https://github.com/espressif/esp-idf/tree/master/examples/protocols/mqtt/tcp

(d) Register this function as the MQTT event dispatcher by calling esp_mqtt_client_register_event with the following arguments: client, ESP_EVENT_ANY_ID, funcnamemqtt_event_handler, and NULL.

(e) Start the client by calling esp_mqtt_client_start with client as parameter.

(f) Verify that mqtt_event_handler is being called when the application starts.

6. ***MQTT Dispatch*** It's time to fill in a proper body for mqtt_event_handler:

   (a) First, create variables for the event data and MQTT client:

   ```
   esp_mqtt_event_handle_t  event  = event_data;
   esp_mqtt_client_handle_t client = event->client;
   ```

   (b) Them cast event_id to a esp_mqtt_event_id_t and switch on the result. Add the following cases:

   - The *default* case which prints out relevant information.
   - MQTT_EVENT_CONNECTED This event is generated when the connection has been established. Here, you should (i) call esp_mqtt_client_subscribe with the client, TOPIC, and the value 1 as arguments, and (ii) create a task called TaskChat. Remember to create the corresponding function for TaskChat.
   - MQTT_EVENT_DATA This event is generated when data arrives on a topic we have subscribed to. It gives us access to the topic and payload like this:

     ```
     printf("TOPIC=%.*s\r\n", event->topic_len, event->topic);
     printf("DATA=%.*s\r\n", event->data_len, event->data);
     ```

     Instead, disregard the topic, and simply print out the data.

7. ***Service Loop*** Fill out the body of TaskChat with a do-forever construct that has the following body:

   (a) Read one line from the serial line.

   (b) Use sprintf to construct a line of the format "%s: %s\n" in a preallocated buffer (you decide the size). The first %s is the nick and the last one is the line you just read. Lets assume that the variable pointing to the buffer is called buffer.

   (c) Call esp_mqtt_client_publish with client, TOPIC, buffer, zero, one, and zero as parameters.

8. ***Test*** Verify correctness. You can use HiveMQ's online client[7] for debugging:

   (a) Press "Connect" to get a session.

   (b) Press "Add New Topic Subscription" and replace the "Topic" text with the topic name from the TOPIC precompiler definition.

   (c) Observe the messages flow through.

   (d) There is also a "Publish" interface where you can choose the values of the topic and message.

[7] http://www.hivemq.com/demos/websocket-client/?