Lab: WiFi

Introduction

The purpose of this lab is to get you some hands-on experience with WiFi on a microcontroller:

- Scan and join networks
- Host a network
- Host and access web services

For this lab we will be writing code using PlatformIO. We'll be working on the Heltec WiFi LoRa 32 v3 boards this time though, which are also supported by the Arduino libraries.

Goals

- Interact with WiFi networks
- Explore some basic IoT web services

Equipment

- Computer
- Heltec WiFi LoRa 32 v3 + USB cable (3 total for the group)

Github Classroom

- https://classroom.github.com/a/gGwmmNMh

Partners

- This lab should be done with **your group of three**

Table of Contents

List of Checkoffs

- Section 5.1: Demonstrate ability to scan networks

- Section 6.1: Demonstrate ability to get time through the Internet

- Section 7.1: Demonstrate ability to capture packets from your other device
- Section 7.2: Determine some information about how the ESP32 client is connected

- Section 8.1: Demonstrate creation of an Access Point

- Section 9.1: What is the **default** throughput you could achieve?
- Section 9.2: What is the **maximum** throughput you could achieve?

- Section 10.1: Demonstrate the full working MQTT app across three devices

Lab Setup

1. Setting up PlatformIO

To program our boards, we're going to use PlatformIO. It's a relatively new embedded toolchain management system. One of the really hard parts about embedded systems development is making sure you have the right tools installed on your computer: compiler, programmer, debugger, etc. And every hardware board and software framework you work with has slightly different tools that are required. PlatformIO automatically determines what stuff you need to program a board, and so far in my experience it does a decent job. It has worked on Windows, Linux, and MacOS (both Intel and ARM). It doesn't have a lot of development progress though, so I'm a little worried about its long-term prospects.

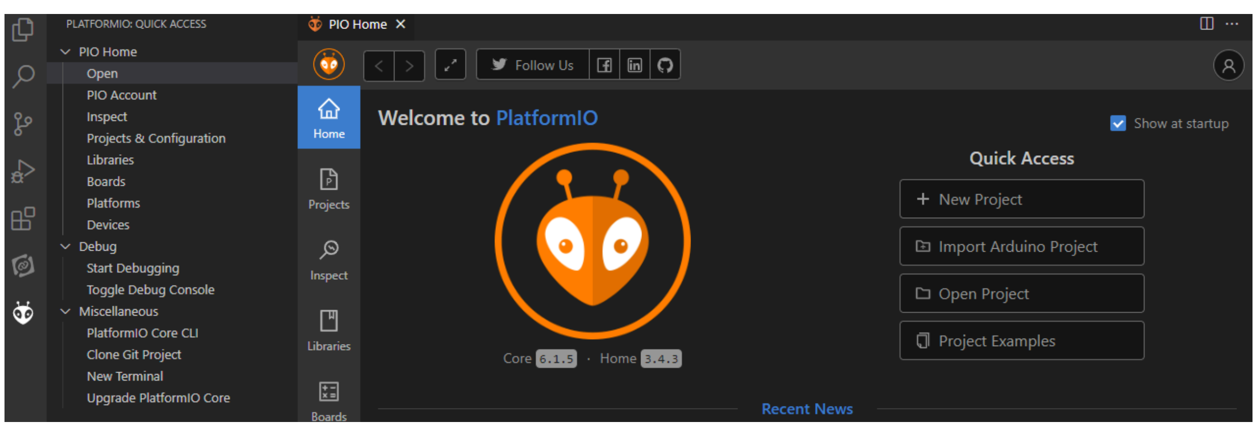
For better and for worse, PlatformIO is almost entirely used as an extension for VSCode, so that means we'll be using VSCode to write our software and PlatformIO to manage our toolchains, compile code, upload it to boards, and display output from boards.

- Install Platform IO

  Follow the steps here: https://platformio.org/install/ide?install=vscode

  It'll take a minute or two to actually install PlatformIO. When it's done, you'll have to reload VSCode and the little Ant icon will appear on the left side of it.

- Get yourself to the PlatformIO homepage



CHECKOFF: None. Continue to the next section.

2. Create Your Lab Git Repo

I'll want to look at the code you wrote, so I need to give you somewhere to put it. Github classroom makes private repos for each student team so you can get the starter code and upload your own modifications. I can access all student repos, but you can only access your own.

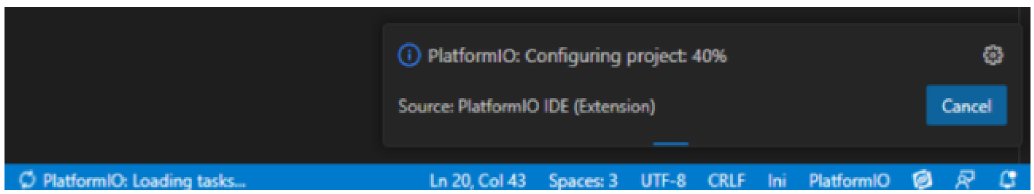
- There is a github classroom link on the first page of this document. Click it!
- Pick a team name
    - Unless someone else already started it, in which case, join their team name
- Generally, do what github classroom says
- At the end, it should create a new private repo that you have access to for your code
    - Be sure to commit your code to this repo often during class!
- The repo link might 404. If so, you first have to go to https://github.com/nu-cs433-student and join the organization
- Clone the repo locally on your computer
    - If you're on Windows: git BASH does a good job https://gitforwindows.org/
        - Make sure there are no space characters in the entire path to the repo. Probably put it somewhere like "C:/iot_apps/REPONAME" if you have a space in your username.
    - If you're on MacOS or Linux, you can clone the repo from command line
    - We'll be using VSCode for everything, and it has a mechanism for working with git too, so you could use that: https://code.visualstudio.com/docs/sourcecontrol/overview#_cloning-a-repository

CHECKOFF: None. Continue to the next section.
However, make sure to commit your code as you go.

3. Load the Example Apps

Let's get an initial app working on the hardware now.

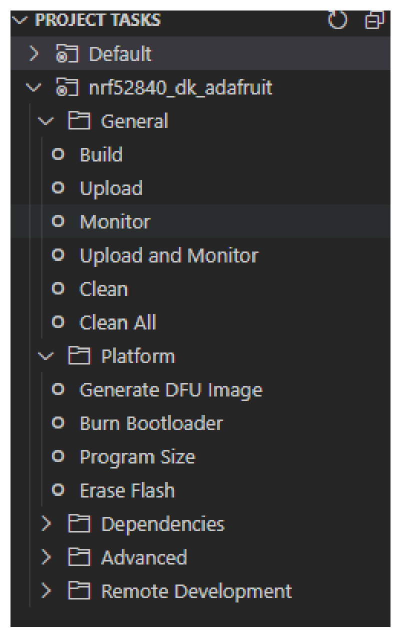
- Use the "Open Project" button on the PlatformIO homepage to open one of the folders within your group's Github starter repo. The best one to start with is "blink-and-print".
    - The first time you open a folder in that repo with PlatformIO, expect a lot of loading to occur. It needs to install an entirely new toolchain to compile and upload code for ESP-32 microcontrollers. On my desktop, this took about five minutes to complete. You'll see something like this:



- You should now have two useful toolbars to interact with code.

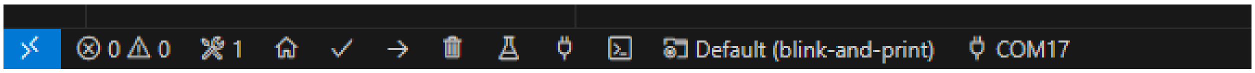
The first is the VSCode explorer tab on the left, which will show you the file structure of the application and allow you to open the code in src/main.cpp.

The second is the PlatformIO tab on the left, which will show you "Project Tasks" like Build (compile code), Upload (load code onto the board), and Monitor (open a serial console to see print output from the board). I mostly use "Upload and Monitor" which recompiles, uploads, and opens a serial terminal all at once. If you later end up opening multiple projects at once, the blue bar at the bottom has both shortcuts to these actions and tells you which project the actions are applying to.



- The bottom banner of VSCode is also useful now. It tells you which application is currently active in PlatformIO and which device you're talking to (Auto by default, but you can pick if you have multiple boards attached).

  Here's my computer when I'm working on the `blink-and-print` application and I'm interacting with the device on COM17 (Windows)



- Plug your board into your computer using your USB micro cable and the USB-C to Micro USB adapter in the box with the board.

- When you plug in the board for the first time, check that orange LED by the "RST" button lights up. That means the board has power.

  A weird quirk of this hardware is that it MUST have a USB adapter somewhere inline when you plug it in. Plugging in directly via a USB-C to USB-C cable fails to power the board (the implemented something about that incorrectly…). Each box should have a USB-C to Micro USB adapter, which you can use in conjunction with the USB micro cable you already have from previous labs.

- In PlatformIO, click "Upload and Monitor" to compile and flash the application and then automatically open a serial port to it.

  - Windows: If you get an error there's a very real chance you need to install a USB driver for the board. It worked fine for me on one of my Windows computers, but not the other two. I needed to install the "CSP210x Windows Drivers" (not the Universal one, actually maybe that one is fine too), and then it worked. https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers?tab=downloads

  - Even after that, you might get a very cryptic error on Windows, about not having access to the COM port. A reboot resolves this issue. Linux and MacOS should hopefully be fine, but let me know if you have issues.

  - I've also sometimes found that commands fail the first time, but succeed the second time. It didn't happen too often, but be willing to try it if things are going wrong.

- The LED should now be blinking (you'll know it's hella bright) and you should see a count of which iteration of the main loop we're on.

- Take a look at the code in `main.cpp` and try modifying some things to make sure you understand it.

- Also try out the `screen-example` application, which demonstrates how to use the screen on the board, which is slow but totally functional.
    - I put a secret in there for you.
    - When you switch apps, if an old serial terminal is still running, programming your board will fail. You'll have to manually close the old serial terminal.

CHECKOFF: None. Continue to the next section.

Reference Documentation

4. ESP32 Arduino WiFi Library

Our Heltec board has an ESP32 microcontroller. It is a general-purpose microcontroller, but it has a WiFi radio peripheral (much like the nRF52 series is a microcontroller + BLE/15.4 radio). We'll be programming that chip and writing our own code in PlatformIO.

Documentation:

- Arduino WiFi Library API:
  https://github.com/arduino-libraries/WiFi/blob/master/docs/api.md
  - WARNING: This isn't actually the library we're using. Our library just emulates this one, and it doesn't match precisely. These docs are way better though for Client-side stuff (they don't have Access Point stuff at all).

- ESP32 WiFi Library Docs:
  https://docs.espressif.com/projects/arduino-esp32/en/latest/api/wifi.html#
  - This **is** the library we're using. But the docs aren't exactly great. You'll still have to use it for Access Point stuff.

- ESP32 WiFi Examples:
  https://github.com/espressif/arduino-esp32/tree/master/libraries/WiFi/examples
  - You'll find that these cover all the basic things you want to do. My strong recommendation is that you don't do too much copy-paste from them and instead use them as inspiration to write your own code. You'll learn more that way.

  - You'll find that a lot of code does get reused across sections of this lab. Definitely copy-paste your own code from prior sections.

For this lab in particular, you are welcome to use code you find online as long as you cite your sources. The goal is to learn about how wireless networks work, with much less focus on the specific code that you write.

CHECKOFF: None. Continue to the next section.

Interact with WiFi

5. Scan for WiFi networks

The first step of doing anything with WiFi is scanning for available WiFi networks.

- Create an app that can scan for WiFi networks and display them in terminal. We've provided some starter code in `wifi-scanner`

- I highly recommend you go look at some example code.

- For each network you find, print out at least the following information:
  - SSID
  - Channel
  - Encryption
  - RSSI

1. **CHECKOFF:** Demonstrate ability to scan networks
   - Show me the terminal output from a scan

6. Connect to the Internet

Now let's connect to the Internet and get some data from it! We'll contact an NTP server and request the current time. It'll be in UTC, but it should make it obvious that everything is working.

- Open the `wifi-client` starter code

- You'll need to specify an SSID and Password for a WiFi network. There are several options here:
  - Option: Use your home WiFi network. Should work fine, but only at home.

  - Option: Use Device-Northwestern network. I got this working fine too! And it works on-campus. But you need to do a setup step.

    You first need to register the device. There's a web portal to do so, and it seems to work automatically and pretty quickly (definitely within a minute). https://device.wireless.northwestern.edu/ (do NOT enable "Sharing".)

    To get your MAC address, you'll have to program an app on your device that will spit out the MAC address as the first thing it does. Make sure you have the endianness right. The device's MAC address should start with Espressif's OUI: F4-12-FA
    - Your device might already be registered from a prior quarter. Turns out IT doesn't remove old devices very often…

  - Option: Use your smartphone as a Wireless Hotspot. I've done this in the past and it was fine. Definitely less desirable, but we aren't going to be transferring any meaningful amount of data.
    - A warning: groups were having problems with apostrophes in SSIDs

- Make sure you can successfully connect to the WiFi network and get an IP address. Sometimes this just doesn't work for me after I reprogram or reset a device. Resetting again once or twice has always fixed it.

- Use the Network Time Protocol to get the current time and date. Here's a library to use: https://github.com/taranais/NTPClient It's already been enabled for you in the application's `platformio.ini` file.

  The example in the README is probably enough to work with. You'll want to print time *and* the date though. And you want to adjust it for local time (there's an offset parameter).

1. **CHECKOFF:** Demonstrate ability to get time through the Internet
    - Show me the terminal output printing the current date AND time in our local timezone

7. Scan Promiscuously

This is a little more off the beaten path, but you can use an ESP32 not just to send packets, but also to receive packets. Most radios, the ESP32 included, have an option called "promiscuous mode" where they collect not just packets intended for them, but **all** packets it can sense. We'll use the ESP32 to do a promiscuous scan of WiFi packets and find packets being sent by the client app we created previously.

- Start from the `wifi-promiscuous` example code.

  The code I set up for you is based on the ESP32 sniffer application here: https://github.com/ESP-EOS/ESP32-WiFi-Sniffer/blob/master/WIFI_SNIFFER_ESP32.ino

- I recommend first getting it to scan any WiFi packets at all to make sure it's working. Print out some details about the packets so you know they seem valid.

- Next, let's find packets being sent by the `wifi-client` app we wrote. You'll need two ESP32s for this. One should use the NTP client app you wrote previously and the other will be running the promiscuous scanner.

  You'll need to figure out what channel your client device is connected on. It's available through the WiFi API once connected. Make the client app print that out.

  In your client app, you should also use `forceUpdate()` for the NTP library so it makes a network request each time instead of caching results for a minute. That way there are more packets to capture.

  To "filter" for only packets from your other ESP32 device, match against one of the address fields. Either `addr1` or `addr2` should contain the MAC address of your ESP32 client. If there's not a match, just return from the scanner function early.

- Once you can find packets from the device, grab interesting data from the packets! This is mostly in the `rx_ctrl` field of the `wifi_promiscuous_pkt_t` struct.

  The field itself is a struct of type `wifi_pkt_rx_ctrl_t`. You can find a full definition here: https://sourcevu.sysprogs.com/espressif/esp-idf/symbols/wifi_pkt_rx_ctrl_t

(Tasks on next page)

1. **CHECKOFF:** Demonstrate ability to capture packets from your other device

   a. Show me the terminal output printing some packet metadata

2. **CHECKOFF:** Determine some information about how the ESP32 client is connected
   a. Which WiFi PHY protocol is it using to connect to the WiFi network? (e.g., 802.11b, 802.11g, 802.11n, etc.)
   b. Which channel is it connected on?
   c. How much bandwidth is it using on that channel, 20 MHz or 40 MHz?

Host WiFi

8. Host a WiFi network and connect to it

Going back to well-worn examples, let's make an ESP32 into an Access Point (AP) that runs its own WiFi network.

- Start with the `wifi-access-point` example code.

- Figure out how to make your ESP32 into a WiFi access point.

  Pick an interesting SSID for your group. You can leave the password parameter empty to make the AP open-access.

- Connect to it with some device.

  A phone works here. Although I find that on Android at least, it automatically disconnects after finding that there is no actual Internet connection.

  You can also connect with your client application! However, it'll find that Internet access isn't available and the NTP library will fail.

- The ESP WiFi libraries can provide more information about clients that are connected to it. Some starter code that fills in a `wifi_sta_list_t` is provided to you.

  You can find the definition for that struct here:
  https://sourcevu.sysprogs.com/espressif/esp-idf/symbols/wifi_sta_list_t

- Print out at least the following information about the clients that are connected:
  - MAC address
  - RSSI
  - Which 802.11 PHY protocols are enabled

1. **CHECKOFF:** Demonstrate creation of an Access Point
   a. Show me the terminal output printing data about a connected client

9. Test Maximum Throughput

What kind of throughput can you get between two ESP32 devices both connected to one ESP32 Access Point? Let's find out!

- Start with the `wifi-client-throughput-sender` and `wifi-client-throughput-receiver` applications. And you'll need the `wifi-access-point` too.

- Set up a UDP server (receiver) and client (sender) to handle one-way data transmission.

  The Arudino WiFi library has a UDP class to make this simpler.
  https://github.com/arduino-libraries/WiFi/blob/master/docs/api.md#udp-class

  - To initialize: call `udp.begin()` in your `setup()` function with a port number to receive on (or zero if you don't need to receive)
  - To send data: `udp.beginPacket()`, then `udp.write()` with an array of data, then `udp.endPacket()`
  - To receive data: `udp.parsePacket()`, then `udp.read()`

  - For the IP address of the receiver, probably start the receiver first, print its IP address after it connects, and then hardcode it in the transmitter.
    - Alternatively, you could have the Access Point print the IP addresses of its clients.
    - Or you could have the router broadcast its IP address via mDNS or UDP broadcast messages. Or you could just ping addresses until something responds (probably only 256 to try?)

- On the sender side, you'll want to send a special "START" packet that's different from the others in some way. This will trigger the sender to start measuring time.

  Further packets can be any byte values you want, but I recommend having the first couple of bytes be a sequence number in case you want to see which packets have arrived or how many packets should have arrived.

  Important: if you're doing a for loop sending packets, you need to occasionally `yield()` to allow system tasks, like servicing the WiFi hardware, to occur. Yielding once per packet sent would make sense. Or alternatively, you can just send a single packet per `loop()` iteration, as `yield()` automatically triggers between `loop()` calls.

- Develop a scheme to detect how much data has arrived in a given time.

  When you receive the "START" packet, you can take an initial timestamp. The `micros()` function is perfect for this:
  https://docs.arduino.cc/language-reference/en/functions/time/micros/

  After that, track how many bytes have been received (maybe including which sequence numbers have arrived). You'll take an ending timestamp and print results after some total amount of data has been received.

  You don't have to ensure that every packet is successfully received, as long as you're accurately counting how many bytes are accumulated from the packets you do actually receive.

- Make sure you don't have any time-consuming function calls in the critical path of sending or receiving. This especially means print statements. Maybe add them to start to debug whether things are working at all, but once things are working, only print before starting or after finishing. Delays will also harm throughput, although that is hope

- Run a test sending at least 10,000 bytes of data from one client to another, determine the time it took to send that data, and print the throughput measured in units of bits per second.

  By default, **each packet sent should have 100 bytes of data**. Don't do anything special to optimize performance of this yet.

1. **CHECKOFF:** What is the **default** throughput you could achieve?
    a. Hang on to this number, as we'll compare against a hopefully better result soon.

- Let's do that again, but try to improve throughput by configuring things better. You'll likely need to make modifications to all of the apps, including the `wifi-access-point`

    - Make sure that your PHY and MAC layer setup are ideal (or ideal as possible anyways).

      You might consider (and this is not an exhaustive list): channel, bandwidth, proximity of devices, transmission power, and WiFi protocol version. There are functions that let you configure and set many of those, although some are meant to be public while [some are internal](). Some may or may not work, so you'll have to play around with it.

      Put a reasonable amount of work into figuring out what might improve throughput, although I'm not going to require you to actually achieve the perfect throughput.

- Make sure that your packet sizes are ideal too. The fewer packets sent, the less overhead spent on headers and waiting for CSMA to transmit, but packets don't get to be infinitely large either.

- Definitely remember not to waste time in `print` or `delay` calls. But `yield()` is still likely necessary.

- Run an optimized test sending at least 10,000 bytes of data from one client to another, determine the time it took to send that data, and print the throughput measured in units of bits per second.

  This time you can use any configurations you want.

2. **CHECKOFF:** What is the **maximum** throughput you could achieve?
   a. What do you think had the biggest impact on the improved throughput?
   b. Show me the code and configurations you used to achieve that

WiFi-MQTT Application

10. Use MQTT across multiple devices

This is the full demonstration application! We'll create a new network and connect two clients to it. On that network, we'll host an MQTT broker on the Access Point, publish from another device, and subscribe from a third device! **The goal is the old classic**: press a button on the publisher and make an LED change state on the subscriber.

If you're feeling iffy on MQTT, here's an overview:
https://learn.sparkfun.com/tutorials/introduction-to-mqtt/all

- Three applications have been created for you:
    - `wifi-access-point-broker`
    - `wifi-client-publisher`
    - `wifi-client-subscriber`

- The library we'll be using for MQTT is TinyMqtt. The documentation is pretty non-existent, but the examples are good. See here:
  https://github.com/hsaturn/TinyMqtt/tree/main/examples

  The `simple-broker` example demonstrates running a broker. The `simple-client` example demonstrates publishing. The `client-without-wifi` example demonstrates subscribing.

  The library has already been enabled for you in the application's `platformio.ini` file.

- The TinyMQTT library requires that you run an update function in your `loop()`. To make this work, you need to avoid any use of `delay()` in your loop. You can still have periodic loop behavior though through the use of the `millis()` function and storing the past time. See our `blink-and-print` code for an example of this.

- Application design: pressing a button on the `wifi-client-publisher` will send an MQTT message. When the `wifi-client-subscriber` receives the MQTT message, it will change the state of its LED. E.g., press the button on one device and make the LED change on the other. You can publish to any topic you want (it's an arbitrary string) and with any data you want (also a string) as long as it makes the behavior occur.

1. **CHECKOFF:** Demonstrate the full working MQTT app across three devices
    - And the code you wrote to make this work