Handheld Wifi Analyzer

Introduction

This project is a handheld Wi-Fi analyzer based on an ESP32 microcontroller. It scans nearby Wi-Fi networks, displays their signal strength, channel usage, and other technical details on a built-in screen. Users can interact with the device using physical controls to navigate and inspect network data.

Its purpose is to be a portable tool for users to understand the wireless environment: troubleshoot issues, find optimal channels, detect signal strength, monitor traffic.

General description

The main component is the ESP32 microcontroller development board with built-in Wifi. The fast processor, low battery use and number of GPIOs make it a good choice for this project.

To show information to the user, this project uses a 2.8” TFT SPI LCD display based on the ST7789 driver. It has a resolution of 480×320 full color pixels. The decent size and resolution makes it a great for display text and graphics.

For user input, I chose a rotary encoder module along with 2 buttons. The encoder makes it faster to move around menus than traditional buttons (scrolling, for example), and thus the 2 buttons I use are for “enter” and “back”.

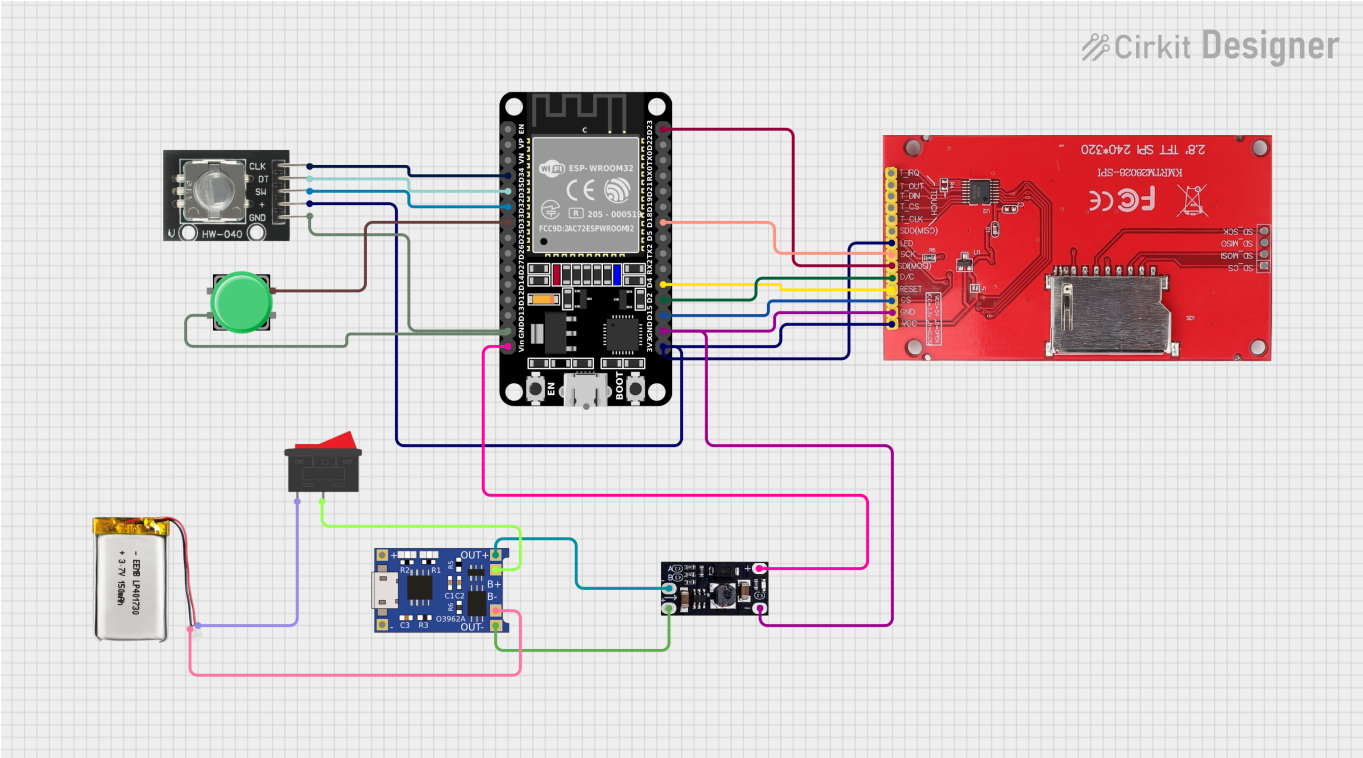
Since this is a handheld device, it uses a battery along with a charging module. The battery is a 750mAh 3.7V LiPo battery connected to a TP4056 charging module with protection. Since the 3.7V are not enough to power the ESP32 and display, I use a 5V 1A boost converter to raise the voltage. USB power is also an alternative to the battery.

On the software overview side, the code is organized into modules, each reporting to the Main module. The Wifi Scanner module handles scanning for networks, reading and parsing packets, storing network and channel info etc. The Battery Info Module is responsible for collecting information about the battery level. The Display Communication Module draws all the information provided by the Main Module to the screen. The Input module monitors changes to the encoder and buttons and reports to Main Module. Logging module just stores some information about the state of the entire application and can be turned off.

Hardware Design

Hardware Used:

- ESP32 with built-in WiFi
- ST7789 2.8” TFT SPI LCD 240xRGBx320
- 750mAh 3.7V LiPo battery
- TP4056 Charging Module
- Miniature 5V 1A boost converter
- KY-40 Encoder Module with button
- 1 push button
- x3 breadboards
- Jumper wires



As seen above in the schematic, here are the pins I used For the display:

- Vcc –> 3.3V
- Gnd –> Gnd
- LED –> 3.3V
- MOSI –> Port 23
- SCK –> Port 18
- CS –> Port 15
- RESET –> Port 4
- DC –> Port 2

The pin mapping does not really matter, as the mapping can be configured in the library used to control the display.

For the encoder:

- CLK –> Port 32
- DT –> Port 35
- + –> 3.3V
- Gnd –> Gnd

For the pushbutton: one side –> Gnd, other side –> Port 33

The battery positive line goes into a switch that disables the battery when using the USB. Then it goes into the TP4056 charging module, who's output is then stepped up to 5V by the miniature boost converter.

Software Design

This project is more focused on software, so there will be a lot of code. Here are the features that must be implemented:

- Scan WiFi networks, store and display their information: SSID, channel, signal strength, authentication protocol, hidden SSID etc.
- Graph channel occupancy: bar chart of how many networks are per channel
- Signal strength graph: display signal strength over time for a particular network
- MAC sniffing: use Promiscuous Mode to capture WiFi Management Frames by channel hopping, record the MAC addresses of all devices around (education purposes only)
- ~~Deauth attack: Perform deauthentication attacks on nearby devices (own devices only)~~ Due to a limitation of ESP32 firmaware, the ESP32 cannot send deauthentication management frames. Instead, this will be a *monitoring* for deauthentication attacks only feature.

To access all of these features, a Main Menu will be provided on the display in the form of a 3×2 grid. Each cell is a selectable button and will lead the user to a new window with the particular feature.

To organize the code, the state of the application will be modeled using a Finite State Machine. For each state, there will be different elements drawn to the screen as well as conditions for transitioning into different states (for example, go back to Main Menu).

For the code, I opted for Arduino IDE. External libraries used :

- TFT_eSPI by Bodmer
- ~~ImGui~~

I really wanted to port ImGui library to embedded, because it is an amazing library and fantastic for this UI use case. It is hardware and OS agnostic, it just requires a way to draw triangles essentially. That means a software renderer drawing into a framebuffer is absolutely possible. The framebuffer is then passed to the TFT library and blit it to the screen. Alas, this ESP32 does not have external PSRAM, so it's not possible to store the entire framebuffer into memory. You don't need to store the entire framebuffer, especially if you write a wrapper on top of the TFT library and call those functions to draw the primitives. Due to time constraints, I gave up trying to port it. But I may come back one day and finish the port!

WiFi Controller

This class is used to scan for WiFi networks. Since the ESP only spends a few milliseconds per channel

to capture access points, every scan will be different: some networks will be missing. Which is why this class caches networks and evicts them if they haven't been seen in more than 30 seconds. The cache is a map with bssid as keys and a custom data type made up of

```
MyNetwork
```

and

```
lastSeen
```

as values.

```
 // 1) Perform active scan (blocking)
int found = WiFi.scanNetworks(
  /*async=*/false,
  /*hidden=*/showHiddenSSIDs,
  /*passive=*/false,
  /*max_ms_per_channel=*/scanTimePerChannelMs
    );
```

```
// write into cache (or update existing)
CacheEntry &entry = networkCache_[net.bssid];
entry.net       = net;
entry.lastSeen  = now;
```

```
// 3) Prune stale entries and build return vector
std::vector<MyNetwork> aggregated;
for (auto it = networkCache_.begin(); it != networkCache_.end();) {
    if (now - it->second.lastSeen > aggregationTimeoutMs_) {
        // drop old
        it = networkCache_.erase(it);
    } else {
        // include in output
        aggregated.push_back(it->second.net);
        ++it;
    }
}
```

Sniffer Controller

This class handles turning on Promiscuous Mode and sniffing all management frames by channel hopping around. It is responsible both for recording the MAC addresses it finds as well as monitoring for deauth frames.

The way the ESP32 firmware handles this is by registering a callback function that will be called for every captured frame. It only provides the buffer, so I must define a custom data type that defines an IEEE 802.11 header and cast to it:

```
typedef struct {
    uint16_t frame_ctrl;
    uint16_t duration_id;
```

```
    uint8_t addr1[6];
    uint8_t addr2[6];
    uint8_t addr3[6];
    uint16_t seq_ctrl;
} __attribute__((packed)) ieee80211_hdr_t;
```

```
esp_wifi_stop();               // tear down any previous mode
delay(100);
wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
esp_wifi_init(&cfg);
esp_wifi_set_storage(WIFI_STORAGE_RAM);
esp_wifi_set_mode(WIFI_MODE_NULL);
esp_wifi_start();
delay(100);

esp_wifi_set_promiscuous_rx_cb(_promiscCallback);
wifi_promiscuous_filter_t flt = {};
flt.filter_mask = WIFI_PROMIS_FILTER_MASK_MGMT;
esp_wifi_set_promiscuous_filter(&flt);

// Enable promiscuous mode
esp_wifi_set_promiscuous(true);
```

I can then retrieve the MAC address and the management frame type like this:

```
// In 802.11 header: skip Frame Control (2), Duration (2), Addr1 (6) =
offset 10
const uint8_t* srcMac = pkt->payload + 10;
char macStr[18];
snprintf(macStr, sizeof(macStr),
       "%02X:%02X:%02X:%02X:%02X:%02X",
       srcMac[0], srcMac[1], srcMac[2],
       srcMac[3], srcMac[4], srcMac[5]);

Serial.printf("MAC: %s  RSSI: %d  CH: %u\n",
              macStr,
              pkt->rx_ctrl.rssi,
              pkt->rx_ctrl.channel);
```

To channel hop, in the loop() function I must run:

```
// Channel hop 1–13 every 200 ms to catch all APs
static uint8_t channel = 1;
esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);
channel = (channel % 13) + 1;
delay(200);
```

UI code

The TFT library only provides primitives and text, so drawing graphs means drawing them from scratch. The DisplayManager class handles the UI based on the state of the application (see Finite

State Machine approach above).

```
void DisplayManager::update(bool dirty) {
  if (currentState_ == lastState_ && !dirty) return;

  switch (currentState_) {
    case State::SplashScreen: drawSplashScreen(); break;
    case State::MainMenu: drawMainMenu(); break;
    case State::NetworkScan: drawNetworkScan(); break;
    case State::ChannelOccupancy: drawChannelOccupancy(); break;
    case State::SignalStrength: drawSignalStrength(); break;
    case State::MacSniffing: drawMacSniffing(); break;
    case State::DeauthMonitoring: drawDeauthMonitoring(); break;
  }
  lastState_ = currentState_;
}
```

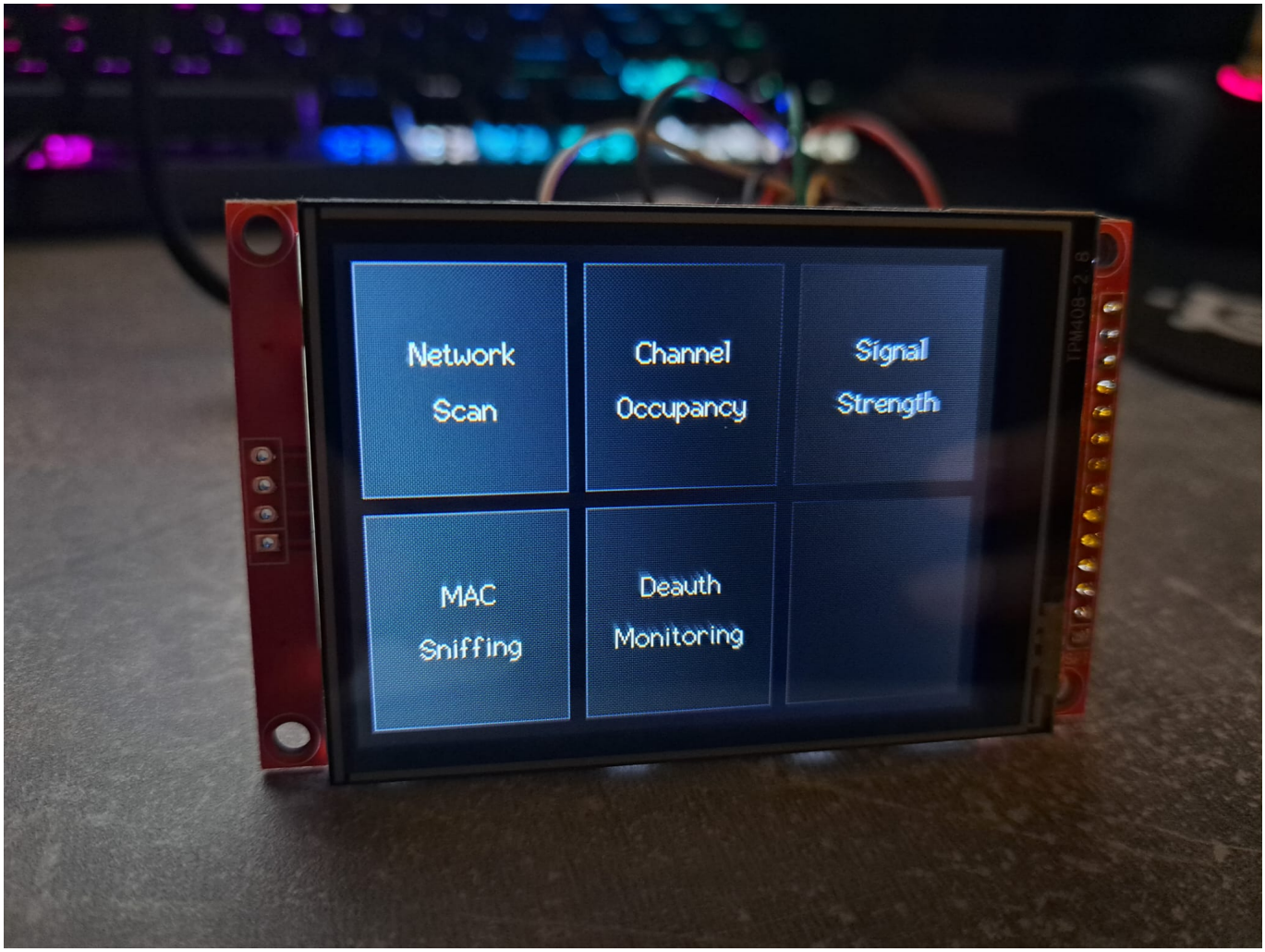
SPI is not fast enough to draw the entire screen continuously without flicker, so only if the state changed or the dirty flag is set manually will (re)rendering occur. Alternatively I can only draw things that will change, such as a graph curve, but keep all labels and axes untouched.

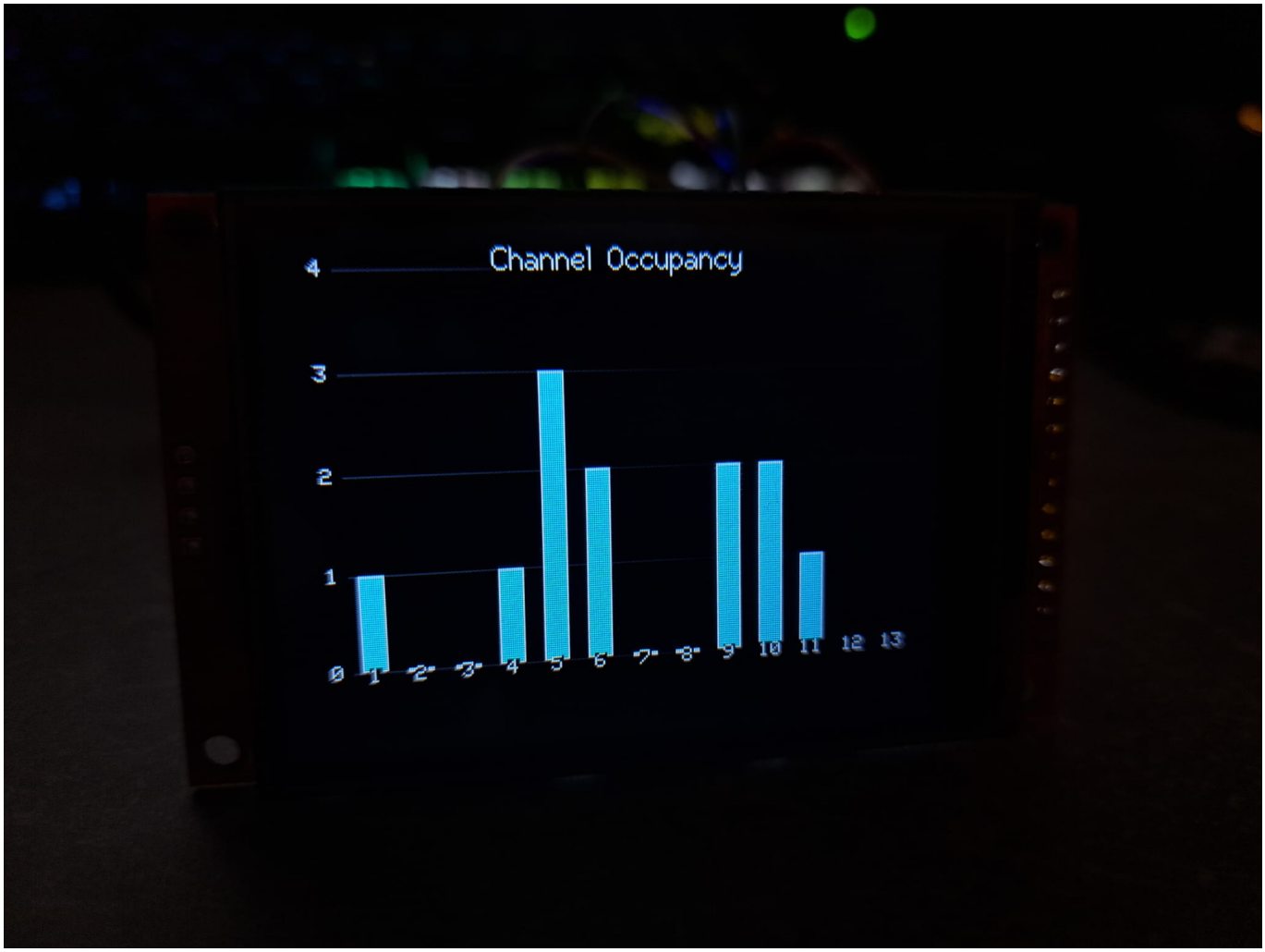
Results

Here are some photos of the UI:


ESP WiFi Analyzer


DIGI-9Xbm -93 dBm CH 12
DIGI-Vkdg -81 dBm CH 11
TP-Link_BE8C -55 dBm CH 9
DIGI-7U42 -89 dBm CH 6
<hidden> -68 dBm CH 10
DIGI-7Uju -82 dBm CH 4
<hidden> -54 dBm CH 9
TP-Link_FEA0 -91 dBm CH 5
<hidden> -91 dBm CH 5
ResidenceAV -68 dBm CH 10


Network
Scan
Channel
Occupancy
Signal
Strength
MAC
Sniffing
Deauth
Monitoring



Concluzii

Download

O arhivă (sau mai multe dacă este cazul) cu fişierele obţinute în urma realizării proiectului: surse, scheme, etc. Un fişier README, un ChangeLog, un script de compilare şi copiere automată pe uC crează întotdeauna o impresie bună .

Fişierele se încarcă pe wiki folosind facilitatea **Add Images or other files**. Namespace-ul în care se încarcă fişierele este de tipul **:pm:prj20??:c?** sau **:pm:prj20??:c?:nume_student** (dacă este cazul). **Exemplu:** Dumitru Alin, 331CC → **:pm:prj2009:cc:dumitru_alin**.

Bibliografie/Resurse

Listă cu documente, datasheet-uri, resurse Internet folosite, eventual grupate pe **Resurse Software** şi **Resurse Hardware**.

Export to PDF