What Plane Is That?

A flight tracking embedded system project.

Reagan

1. Intro

This morning I saw a cool video about a group of guys writing a program that pulled data from flight radars, and displayed it on a screen by their apartment window as a plane flew by. I also live near an airport, and decided to tackle a project like this since I love hands-on projects that you can use as decoration. However, I am strapped for cash at the moment, and I am moving soon, so I will not live near an airport once summer ends… Regardless, here's a step by step demonstration of how this project can be set up, hopefully enabling *you* the reader with the information needed to make your own flight info display.

First, we must discuss hardware. We need to display flight data on some kind of screen, and I decided to go with the Adafruit 32x32 LED Matrix Panel (figure 1). This is $30 and looks decent, with 1,024 LEDs we also have plenty of real estate. This thing, as expected, requires 13 GPIO (general purpose input/output) pins to control it, plus 20 Watts of power (4 Amps at 5 Volts). For the microcontroller, I am a big fan of the ESP32 due to its cheap price, extensive documentation, decent processing power, and ability to easily program with C++. Luckily, the documentation for the LED matrix suggests an Adafruit Matrix Portal S3 board (figure 2), and it is ESP32 based with plenty of memory and processing power. A regular ESP32 development board would be fine, but it would be a pain to attach all 13 pins from the LED board, whereas the Matrix Portal S3 comes with a nice adapter, is $20, USB-C powered, and uses the ESP32-S3 Wi-Fi+BLE chipset.

Second, we need to somehow get flight data for only the flights we can see out of the window. We can't just hop on www.flightradar24.com and find a flight that passes by, we need to get the data in a simple format that we can toss into the LEDs. My thoughts as of now are to display four lines on the LED panel - first line is the source airport three letter code, second line is a down arrow icon, third line is the destination airport three letter code, and fourth line is extra info as desired. To get this information, we will need some sort of free API to request data on flights in the air. OpenSky Network API looks like it will work for this. They use a REST API which is very simple, and an example in their documentation indicates we can filter by latitude and longitude.

Third and finally, we must figure out what we can see out of the window. I am not using my own address for this example but regardless we need to realize that radars show flights on a map top-down. If I choose to filter flights flying over my address, I would not see it out of my window since I would have to look directly up. Instead, we must figure out where on the map we are seeing when looking out at a 20-45 degree angle (likely the angle we would see out of a window, as measured by my level). My window has a ton of trees outside so the view is pretty restricted. We will get more into this later, but I drew a picture of this issue in figure 3.

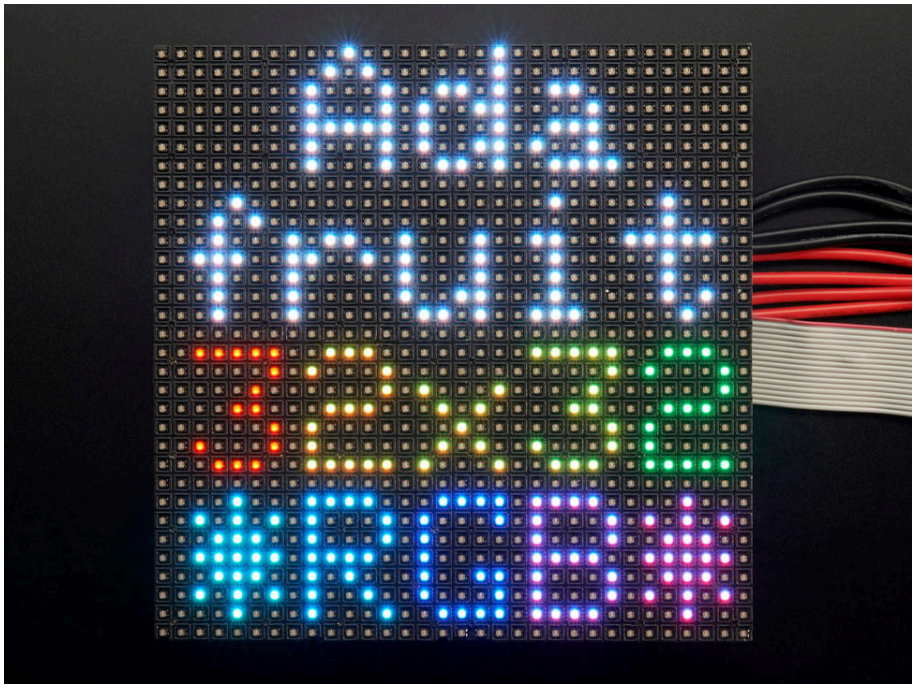


Figure 1 - LEDs

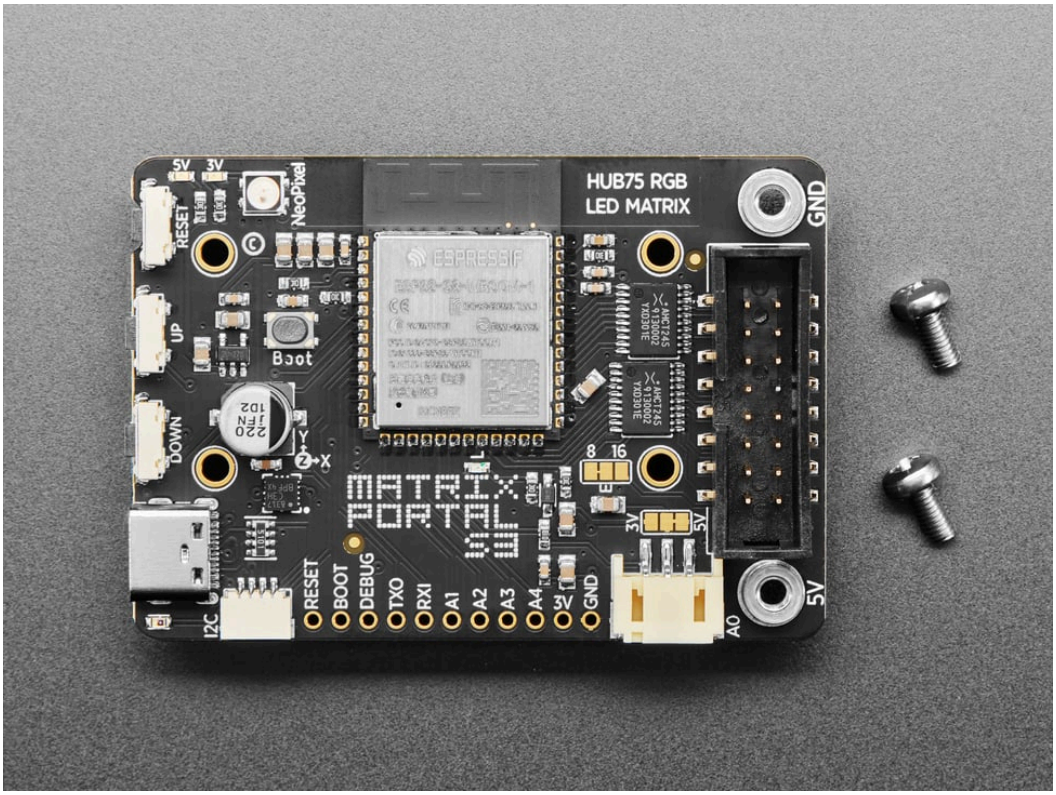


Figure 2 - Board

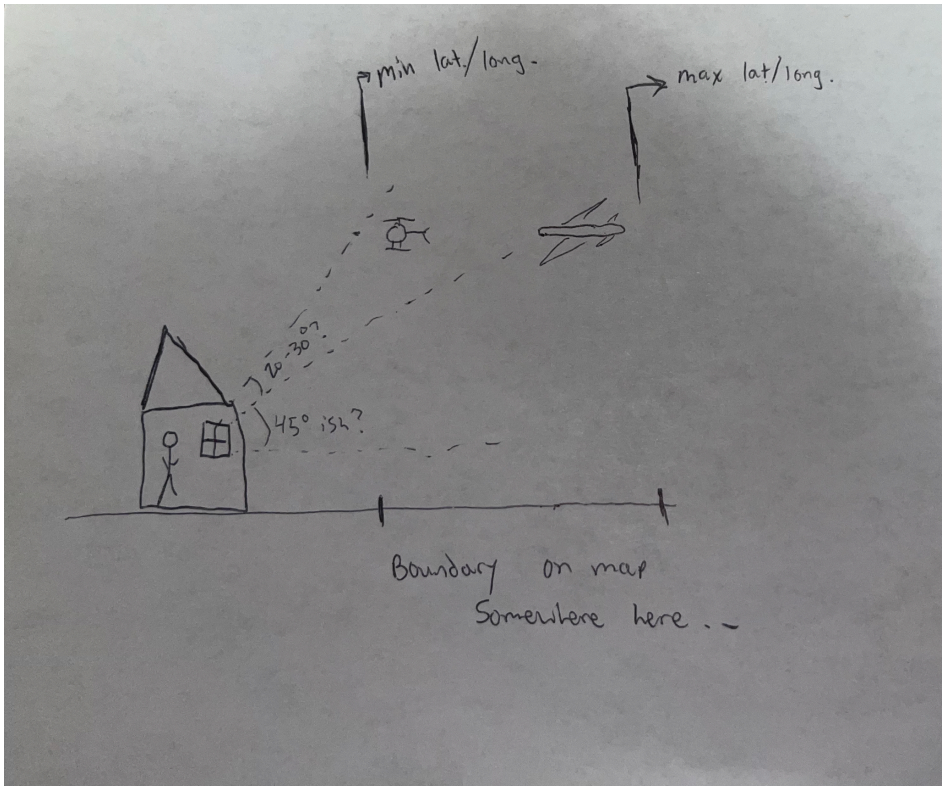


Figure 3 - View

2. The View Outside From My Window

We must figure out the spot on the map we are seeing when looking into the sky from the window. Imagine some infinitely long sticks placed on the ground in a box that mark out the field of view we see when looking out at a 45 ish degree angle. The location of those sticks on the ground will be a ways out from your address, and those are the bounds of our latitude and longitude for grabbing flight data. Anything involving angles is likely going to use trigonometry and triangles, and that is exactly what I did. The farthest out we would likely see a plane is at 45 degree angle with the plane at cruising altitude (~ 10 km). With this info, we can calculate the view distance and the map distance (as shown in figure 4). This is likely going to be obscured by clouds, but this is just a personal project, so tracking cloud radar is not going to be discussed. Given 45 degrees and 10,000 meter altitude, our view distance comes out to be ~14 km. This is way too far, the plane would be a dot in the sky, but with this observation we can change our approach, pun intended.

Let us exclude planes that would be too small in the sky, say, less than the size of a pinky nail at arms length. This is a classic astronomy calculation of angular size. Assuming the average pinky nail is 9 mm wide, arms length is roughly 75 cm: $physical\ size \div distance = (9/1000)\ meters \div (75/100)\ meters = 0.012\ radians$. This is the minimum size of plane we want to grab. Average big commercial planes are ~60-75 meters, so I will choose 70 meters for an estimate. We now have an angle and a side length of a really small triangle shown in figure 5, and solving for the hypotenuse yields a distance of: 5833 *meters*. Rounding to 5km for simplicity, we now have our max view distance, and therefore can find our max map distance. The minimum distance can be found assuming the final approach altitude of planes is 500 feet (~150 meters) and our minimum angle is 20 degrees. See figure 6 for my results.

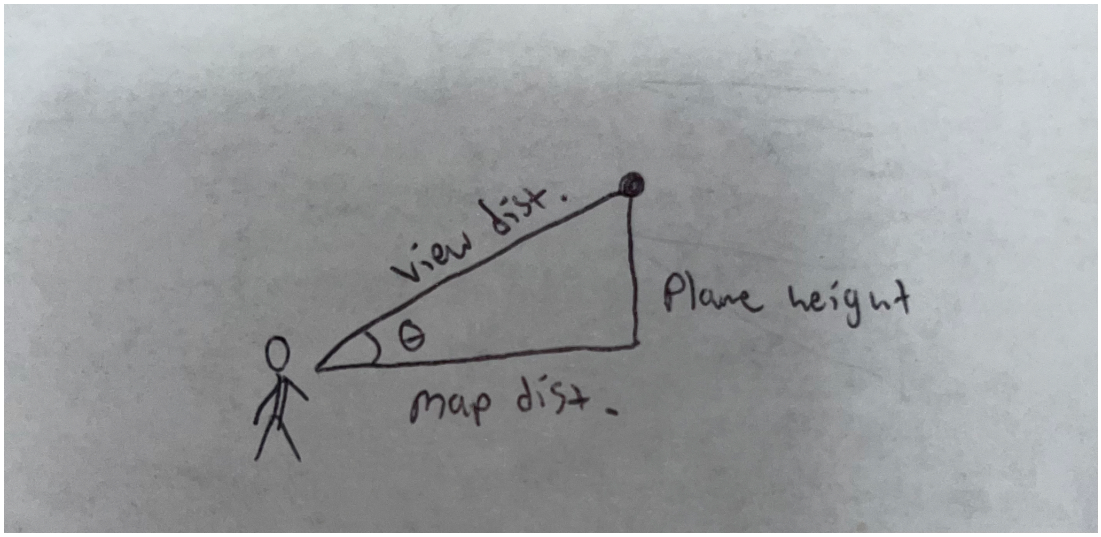


Figure 4 - Triangle

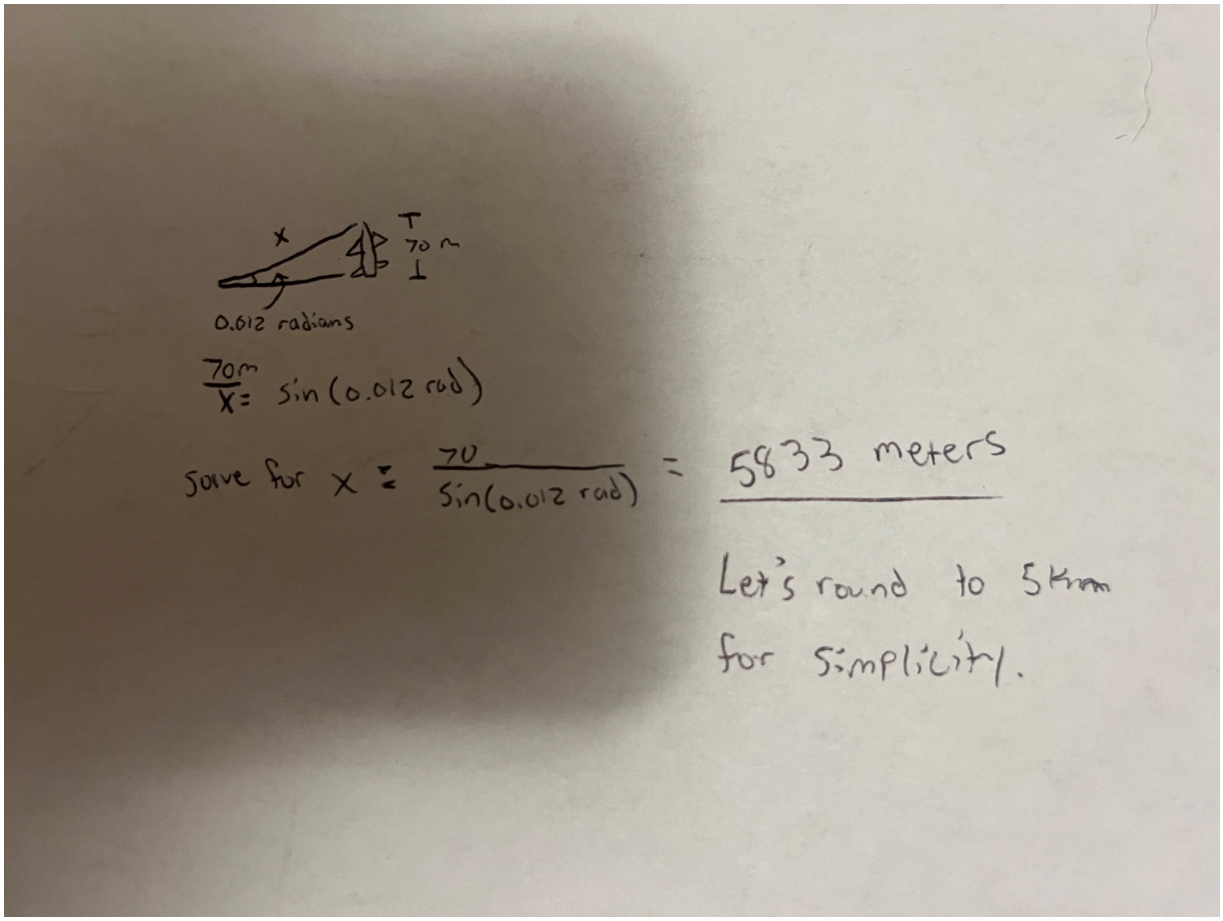


Figure 5 - Max View Distance

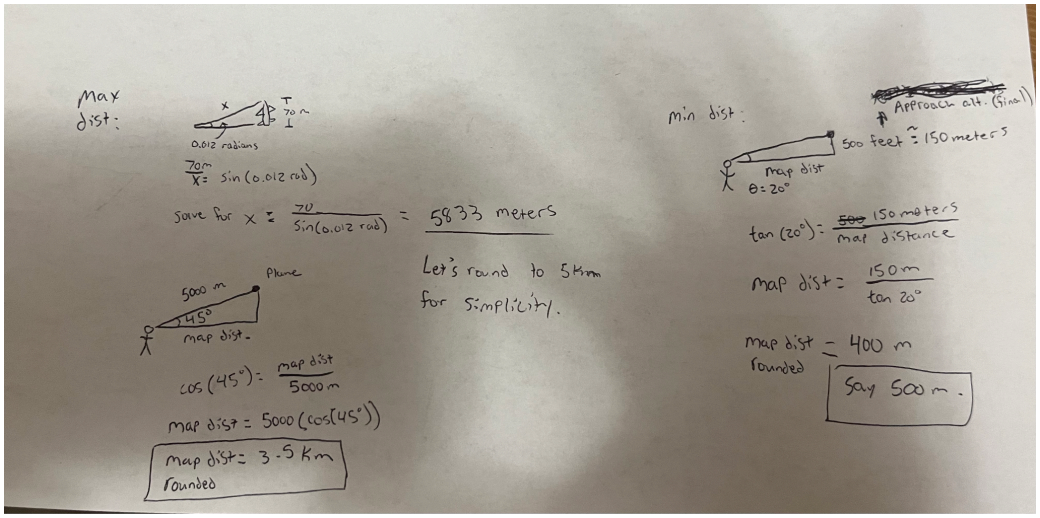


Figure 6 - Min and Max Map Distances

3. Getting Flight Data

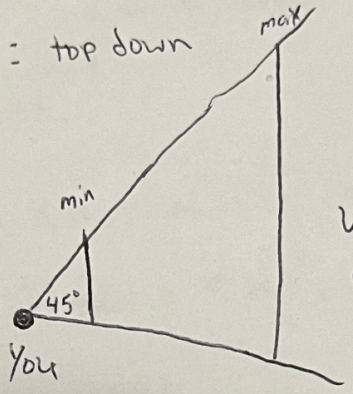
Now that we have our min and max map distances of planes in our area (500 meters and 3.5 km), we need to filter latitude and longitude. The OpenSky API allows us to filter by min/max lat/long (example query is https://opensky-network.org/api/states/all?lamin=45.8389&lomin=5.9962&lamax=47.8229&lomax=10.5226). Lamin, lamax, lomin, lomax are our boundary points. However, we need to figure out where these are on the map. In the section above we found the map distance of planes, but only in a straight line. Assuming the horizontal field of view is 45 degrees, we can use more geometry to figure out how far out from the user's address we need to place our boundary points (figure 7). The results of this depend on the window's cardinal direction. Assuming it faces east, the boundary box will start 500 meters east of the user's address, 200 meters north and 200 meters south for the left side, 1500 meters north and south for the right side, with 3000 meters between the two, as seen in figure 7. It creates a trapezoid on the map.

Min map dist = 500 m

max = 3500 m

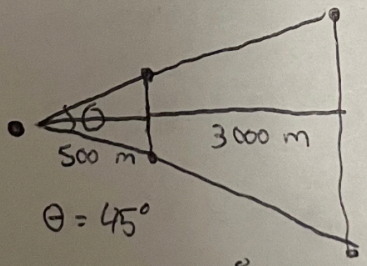
Horizontal view angle: 45°

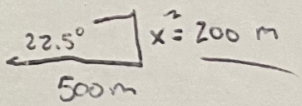
map view: top down

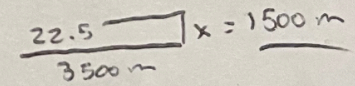


API requires min & max Lat. & long.

So it'll make a quadrilateral on map.

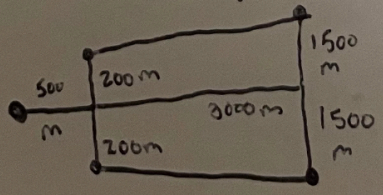






not sure on window's exact direction.

we can make it expandable by generalizing with a ~~rectangle~~ quad



← map mask

Figure 7 - Map Mask

This process can be set up with a program that takes in the user's address and the direction their window faces (N, NE, E, SE, S, SW, W, NW). My first idea was to program a function to calculate four trapezoid points for each cardinal direction and just add those to the coordinates of the given address. This is tough to do for NE, SE, SW, NW, and varies based on hemisphere. This also will not work at the poles (but will this project be used there? No). My second idea was to draw a mask on a GUI that overlays on the map and have the user click and drag it over their address, but that's too complex. At the end of the day, I am going to manually grab these coordinates on google maps. I live close to the DFW airport, so I will use that as my base, but will choose a random location in the city rather than my house for this demonstration.

The Toyota Music Factory is the location I chose for a hypothetical demo, it roughly faces the airport and is close by. After lots of painful line drawings on google maps, I have the rough latitudes and longitudes of my trapezoid: 32.87627, -96.94915 ; 32.87272, -96.95069 ; 32.89334, -96.97684 ; 32.86698, -96.98489. Using the example query from the API docs, the minimum/maximum latitudes/longitudes, we get the link: https://opensky-network.org/api/states/all?lamin=32.86698&lomin=-96.98489&lamax=32.89334&lomax=-96.94915.

Not many planes in the air right now, so the link is returning NULL. It does work though, the example link with a bounding box in Switzerland is showing flight numbers of planes over the country right now. We are good to move onto how the data comes in from this GET request link. In a REST API, a GET request simply asks a server for information. We want flight data that our custom link provides, but need to know how it comes in. The documentation says it returns a ton of stuff in JSON format: callsign, origin country, velocity, altitude, if it is grounded, plane type, etc. Going to the link in a web browser shows us all the data it reads, shown in figure 8. All we need to do is parse the JSON string and extract only the data we want. Since we said our LED matrix will have 4 lines, line 1 will be the origin country, line 2 will be the arrow icon, line 3 will be the call sign, and line 4 will be the velocity (m/s). Some stuff can come in as NULL, which is fine, we will just leave it blank or put some random icon.


```
time:      1751329386
▼ states:
  ▼ 0:
      0:   "40665f"
      1:   "TOM51T  "
      2:   "United Kingdom"
      3:   1751329385
      4:   1751329386
      5:   8.8853
      6:   46.9875
      7:   10972.8
      8:   false
      9:   219.07
      10:  300.48
      11:  0
      12:  null
      13:  11544.3
      14:  "7511"
      15:  false
      16:  0
```


Figure 8 - JSON

From the documentation, we see that it returns a timestamp integer and a state array. Each item in the state array corresponds to the 0-16 you see in figure 7. We need to only grab indexes 1, 2 and 9 for origin country, callsign, and velocity. This can be done manually with memory allocating the whole JSON array and parsing the string at a given index until we hit a null terminator, or we can just use a JSON library to easily grab what we need. Once we store the correct data from the flight trackers, we simply output it to the LEDs.

4. LEDs

The 32x32 LED matrix luckily comes with a library that handles the heavy lifting for us thanks to Adafruit. Trying to update the LEDs manually would be painful. First we would need to scroll through row by row hundreds of times per second and light up the LEDs based on what text needs to be displayed. To figure out what text to show, you would have to manually create a font for every letter in the alphabet and then calculate offsets for how to space the letters on the 32x32 grid, as roughly demonstrated in figure 9. The JSON library also helps clean stuff up, and referring to the documentation we can know what length of data to expect to come in from the API. From here on out it's just programming, fetching the data from our custom link and updating the display every couple of minutes. The code is shown below and also included in the Git repo.

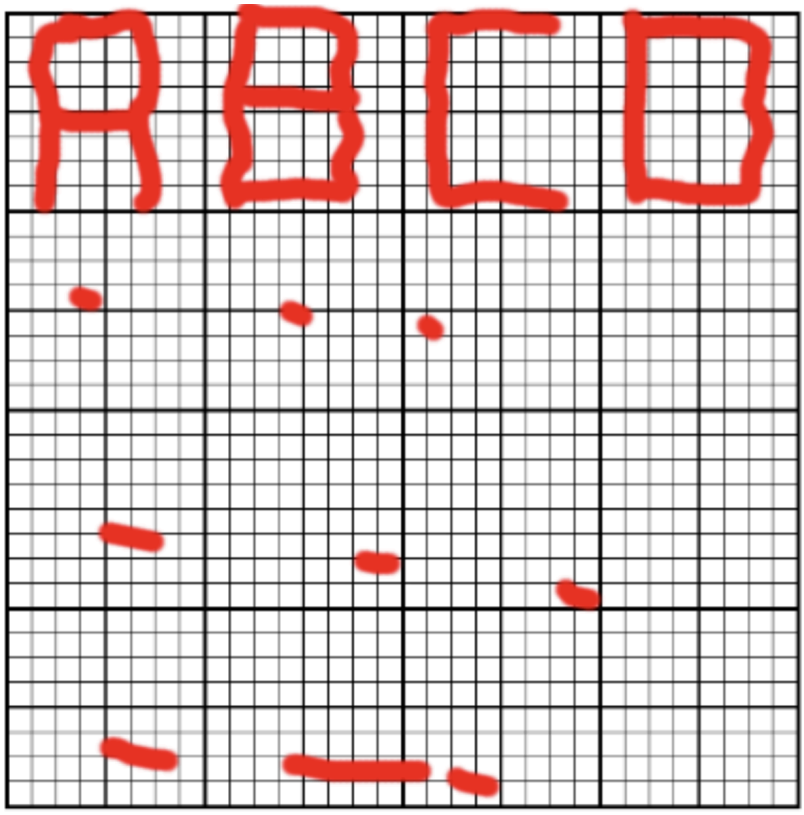


Figure 9 - LED Matrix Font Grid Example

I recommend using the Arduino IDE since you can easily look up how to connect ESP based boards to the program and it will flash code to the chip with a click of a button. It also has a library manager that allows easy installation of the libraries you'll need to run the code. You need to install the HTTPClient, ArduinoJson, Adafruit_Protomatter (LED matrix driver), and Adafruit_GFX (graphics / font system) libraries (as seen in the #includes of the program). From there I suggest reading the code to see how it simply grabs what we need and prints it to the LEDs nicely. I would go deeper into this, but I gave myself a day to document this project and, again, I have no money and therefore no hardware to test this - so it may not work on the first try. But there you have it, the project to get flight data from the sky. This was fun to work through, and the math was simple but fun to work out from scratch on paper.

C++:

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
#include <Adafruit_Protomatter.h>
#include <Adafruit_GFX.h>

#define WIDTH 32
#define HEIGHT 32

const char* ssid = "your_ssid";
const char* password = "your_password";

// Our hypothetical URL for DFW view
const char* apiUrl =
"https://opensky-network.org/api/states/all?lamin=32.86698&lomin=-96.98489&lamax=32.89334&lomax=-9
6.94915";

// RGB Matrix config (Matrix Portal S3 defaults)
int rgbpins[] = { 7, 8, 9, 10, 11, 12 };
Adafruit_Protomatter matrix(
  WIDTH, 4, 1, rgbpins, 6, 2, 3, 4, 5); // width, bitDepth, numPanels, rgbpins, addr, latch, OE,
clock, etc.

void updateFlightDisplay() {
  HTTPClient http;
  http.begin(apiUrl);
  int httpCode = http.GET();

  if (httpCode > 0) {
      String payload = http.getString();
      Serial.println("Received JSON:");
      Serial.println(payload);

      // Allocate JSON document (20 KB should be PLENTY. Really only if your bounding box is near
the airport and tons of planes are around)
      DynamicJsonDocument doc(20000);
      DeserializationError error = deserializeJson(doc, payload); // error checking

      if (!error) {
      JsonArray states = doc["states"];
      if (!states.isNull() && states.size() > 0) {
      JsonArray plane = states[0];  // take first plane

      String callsign = plane[1] | "-----";
      String country = plane[2] | "---";
      float velocity = plane[9] | 0.0;

      // Clean up the text
      callsign.trim();
      country.trim();
      int vel = round(velocity);

      // Display
      matrix.fillScreen(0);
      matrix.setCursor(0, 0);
      matrix.print(callsign);
      matrix.setCursor(0, 8);
```

```
      matrix.print("v="); matrix.print(vel);
      matrix.setCursor(0, 16);
      matrix.print(country);
      matrix.show();
      } else {
      Serial.println("No states found");
      }
      } else {
      Serial.print("JSON parse error: ");
      Serial.println(error.c_str());
      }
  } else {
      Serial.print("HTTP error: ");
      Serial.println(httpCode);
  }

  http.end();
}

void setup() {
  Serial.begin(115200);

  // WiFi setup
  WiFi.begin(ssid, password);
  Serial.print("Connecting to WiFi...");
  while (WiFi.status() != WL_CONNECTED) {
      delay(500);
      Serial.print(".");
  }
  Serial.println(" Connected!");

  // Matrix setup
  matrix.begin();
  matrix.fillScreen(0);
  matrix.setTextColor(matrix.color565(255, 255, 255));

  // Get flight data
  updateFlightDisplay();
}

void loop() {
  delay(2 * 60 * 1000); // 2 min delay between updates, but change as you wish. This freezes the
microcontroller though. Not a problem for us.
  updateFlightDisplay();
}
```

Again, I didn't compile this code, expect errors! For something you could actually use, here's a python program you can run on your machine that grabs the flights around the DFW metro area, also attached to the github repo (use latest python version and have the requests library installed):

PYTHON3.11:

```
import requests

def get_flights_in_bbox(lamin, lomin, lamax, lomax):
      url =
(f"https://opensky-network.org/api/states/all?lamin={lamin}&lomin={lomin}&lamax={lamax}&lomax={lom
ax}")
      print(f"Fetching flights over area: {lamin}, {lomin}, {lamax}, {lomax}")
      print("URL: ", url)

      response = requests.get(url)
      if response.status_code != 200:
      print(f"Error fetching data: HTTP {response.status_code}")
      print(response.text)
      return

      data = response.json()
      states = data.get("states", [])
      if not states:
      print("No flights found in this area.")
      return

      print(f"Found {len(states)} flights:\n")
      for flight in states:
      callsign = (flight[1] or "").strip() or "N/A"
      origin_country = flight[2] or "N/A"

      velocity = flight[9]
      if velocity is None:
            velocity_str = "N/A"
      else:
            velocity_str = f"{velocity:.1f} m/s"

      altitude = flight[7]
      if altitude is None:
            altitude_str = "N/A"
      else:
            altitude_str = f"{altitude:.0f} m"

      print(f"✈️ {callsign:10} | {origin_country:20} | Vel: {velocity_str:>7} | Alt:
{altitude_str:>7}")
      print()

if __name__ == "__main__":
      lamin = 32.6
      lomin = -97.0
      lamax = 33.2
      lomax = -96.4
      # these are for DFW general area, google the bounding box coordinates around your city as
desired.

      get_flights_in_bbox(lamin, lomin, lamax, lomax)
```