SSODD

Sixth Sense Omnidirectional Drone Detection

Will Covington, Dominic Keene, Josh Lehman, Danny Mullen

2024 Notre Dame Electrical Engineering Senior Design

Final Report

Table of Contents:

1 Introduction

Small Unmanned Aerial Vehicles (UAV) are becoming increasingly present on the modern battlefield. This growth in drone usage has become especially apparent in the war in Ukraine. The sophistication of these drones varies from coordinated swarms with integrated sensors and communication systems to hobby drones, explosives, and duct tape. Drones can be used for a multitude of purposes, from surveillance to explosive elimination. In the case of the latter, a three second warning could be the difference between the life and death of a soldier.

There is no current method for our armed forces to detect incoming drones. Special Operations Command has tasked us with creating a wearable device that could alert a soldier to the presence of an incoming UAV.

A detection device would have to be quick, reliable, and unobtrusive. Just like the detection of conventional ordnance, every millisecond counts. The sooner the user can receive the alert, the more time he or she has to react. False positives from the device would slow down operations and wear down the user's trust in the readings, while false negatives could result in injury or death of the user. The device should not interfere with the intended force's range of motion or add significant payload to their already significant fighting loads.

Our design solution has two drone detection methods, through WiFi sniffing and audio frequency spectrum analysis. These two methods combine to determine whether what they detect is actually the presence of an unidentified drone. This determination is then communicated to any other interconnected systems which send an analog output if there is a drone detected. The final design includes the basic components that we expected, working more or less as we expected, from the beginning of the semester. The specifics of each component and subsystem do look substantially different, however.

First, we expected the RF detection method to be a breakout module which would scan for the presence of a drone, using SPI. We looked at many options and eventually found the ESP32 WiFi module to be a sufficient alternative. The software for this subsystem also looks different than expected. In fact, it is the opposite of what we originally expected it to do. At the beginning of the semester, we thought we were determining what an incoming UAV is. Now, our final design actually detects what a UAV *isn't*. The device must be trained for each location of use, and then sort detected WiFi signals through a table of knowns, such that an unknown can be determined as a drone.

The audio sensor system looks most similar to what we expected it to be. We'd originally intended to use larger directional microphones, but opted for a small electric microphone which would fit on our board design and work sufficiently to prototype a sound detection system.

Communication between boards looks nearly identical, as we expected to use ESP-NOW as our protocol. One major difference from our expected design of the network, however, is that we expected there to be one central board and many satellite boards which would be a simpler design. In the end, we opted to make each board identical. They could be wired up differently such that one works solely as a detector and the other solely as an output (as shown in our demo). However, the great benefit to this design is the flexibility. There could be a hub with many satellites, or if each board is equipped with all of its capabilities, they could be programmed to communicate as a mesh network. In the future, it could be worthwhile to test each networking method to see what makes sense in each individual use case.

The subsystem that looks the most different from our proposed system is the sensory output. A major part of our original proposal was the use of a TENS unit for a “sixth sense” stimulus output. We’d also considered a wearable wrist watch device which would vibrate as a warning stimulus. In our final design, however, we used a simple analog output with a vibrating motor as the stimulus. From the first time we presented our proposal, there was some backlash against our idea of a TENS unit as the output. Concerns ranged from how sweating and active movement could impact the TENS staying properly adhered to the user’s body to the safety of the unit as a whole, potentially leading to shock or muscle paralysis if used or programmed incorrectly. Additionally, we had significant concerns about the use of a wearable device on the wrist, as such devices are so ubiquitous that it is almost impossible to distinguish one buzzing sensation from another. Regardless, we were intrigued by the TENS unit idea and moved forward with it into the semester. After buying and testing an off-the-shelf TENS unit, we took it apart and tried to analyze the circuit. It quickly became clear that we would have to create a similar device from scratch, and after some research it was apparent that the idea could not be achieved safely in the context of this project. From there we moved forward with a generic analog output so that it could be weighted, and eventually decided on the vibrating motor to still give a tactile sensation (without electrocuting any professors who would examine the demonstration).

It is worth noting that a team member had the chance to speak with some of the possible end-users of the product. They confirmed our notion that a wearable wrist device would be rendered useless by the sheer number already in circulation. They let us know about a possible place for our analog output in the heads-up display of developing eye protection technology. We could not interface with the tech in the timeline of this project, but the analog output we designed could lead to simple implementation in such a device.

In general, our final demonstrated project answers many of the problems we looked to address, despite the winding paths of design along the way. As in any project, the design process is not linear, and changes in our plans were to be expected. We recognize the limitations of the device which will be discussed in later sections, but also look at our project as a first working prototype of a design which could one day see end use after some iteration.

2 Detailed System Requirements

In order to describe requirements to solve a problem, we must first examine the problem being addressed. On the modern battlefield, drones are becoming increasingly more common as a threat for random attacks. Commercial off the shelf drones are relatively cheap, and can easily be equipped with explosives and driven into groups of soldiers. With ever-increasing drone flight speed, timing is everything in determining potential danger. While disruption of drone control is one solution to this, the problem we are addressing is alerting soldiers to the presence of a harmful UAV with enough time to find safety before an attack occurs.

In general, for our design to be successful, it must be fast, reliable, and unobtrusive. The earlier a drone can be detected and that information can be communicated, the faster a user can find safety, greatly reducing the likelihood of casualty. Reliability is crucial in a twofold manner. An overwhelming number of false positives would slow down operation of both the device itself and any military unit which may be using it. This could also create a lack of trust in the system in a “boy who cried wolf” scenario, where the user may learn to ignore warnings and eventually ignore the warning of a legitimate threat. On the other hand of reliability, false negatives would render the device useless, not performing its basic function of drone detection. One priority in the design is making it as small and unobtrusive as possible. Soldiers are already carrying significant loads, and this device should increase safety without being overbearing. A prototype will prioritize smaller devices and a simple board design with minimal surface area.

Additionally, there are some more specific requirements for this particular design which will be necessary for success. These can add on to or partially address the above general design requirements.

First, our system should include multiple types of redundancy. This could be in the form of multiple boards working together in parallel, such that if one fails and the other detects, both output the same signal. It could also be in the form of multiple detection methods. We chose to address this by having both WiFi sniffing and audio frequency spectrum analysis as detection methods, inherently increasing the reliability and usability of the overall system.

The system must be able to communicate with other similar systems. This could look differently depending on the use case, but some ideas include a spoke and wheel design, with one central sensing unit and multiple "satellite" units which give a stimulus output upon receiving a signal from the central unit. Alternatively, a mesh network of many sensing units, each with sensory output and communication capabilities, would satisfy this design requirement.

While not necessarily addressing the potential final product, the nature of our prototype which will be presented for senior design is that we should design a modular system. The board design should align with this such that any mistake in one section will not ruin all other subsystems, and different boards could be connected as needed. In the context of a final product, a modular design will include the ability to act as either the mesh or hub and satellite network as described above, depending on the software implementation.

The desire of the project is to work towards a sensory stimulus output, weighted depending on the detected and calculated threat level. This leads to a requirement for a reliable, analog hardware output which could be connected in a multitude of ways. Originally, the idea was to use a TENS or related system as a tactile sensory output. With a generic analog output design, this could easily be translated from tactile to visual or audio output as desired.

For end use, our design should be battery powered, compatible with portable packs currently used in industry. With that being said, there may be limited power access in end use cases, and the design should minimize power consumption as much as possible. We recognize that future iterations of the prototype could continuously address more energy-efficient options.

3 Detailed project description

3.1 System theory of operation

The system works largely as expected and designed throughout the semester. Two detection systems come together in an algorithm to produce an output which is then communicated between boards.

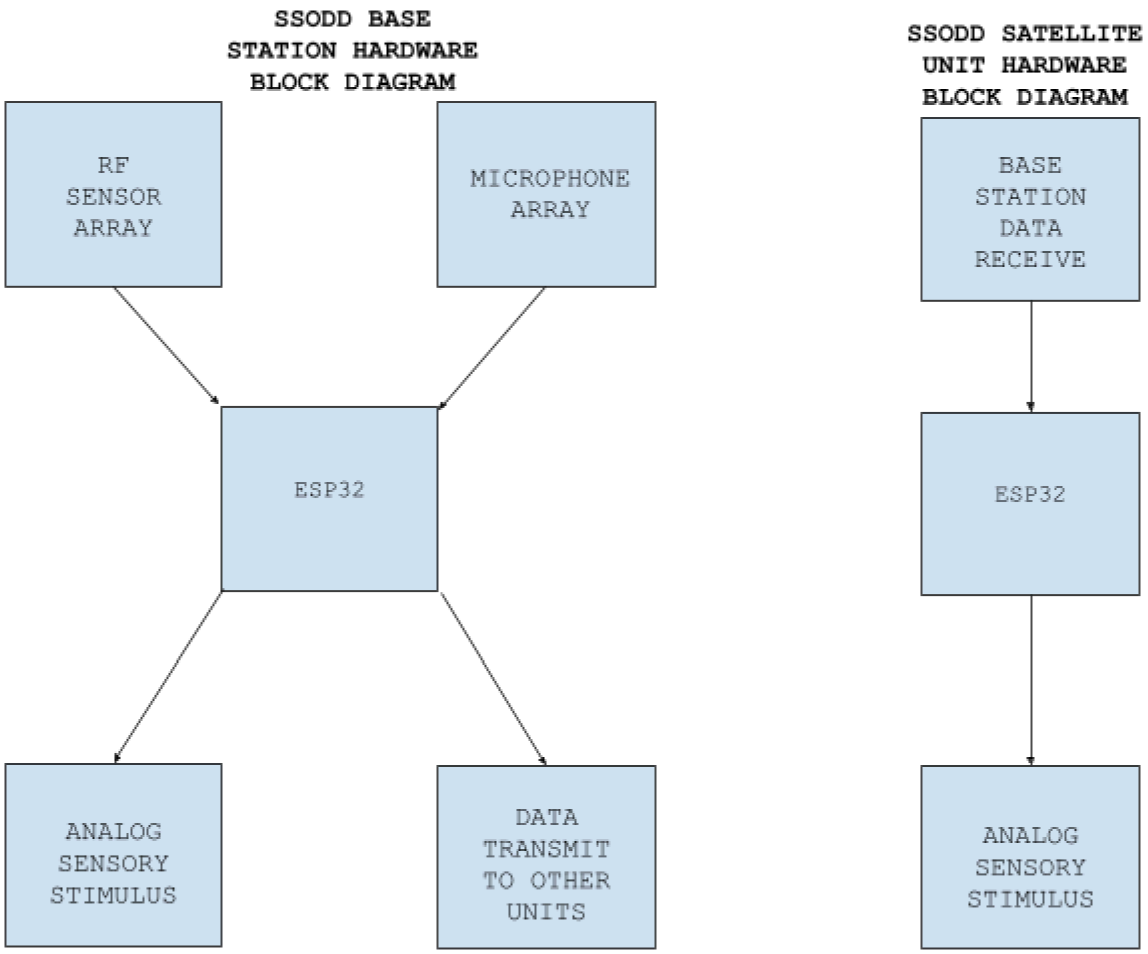
The first detection system uses WiFi sniffing in the 2.4GHz range to determine potential threats. The hardware in this system is based entirely on the WiFi compatibility of the ESP32. This system is constantly reading local MAC addresses and comparing the UOIs (first 3 bits) of each with known addresses in a lookup table. The device would need to be trained for each location of use, adding known addresses to the table. This method, rather than searching for the drone, actually sees what it knows is *not* a drone. From there, unknown addresses that pass through the lookup can be identified as a threat (or at least some unknown which should be investigated).

The second detection system is a frequency spectrum analyzer. The only piece of hardware is a small, electric microphone which outputs a frequency spectrum of all audio it picks up. This passes through a designed analyzer which determines if detected frequencies in the

100-300 Hz range are above a certain threshold defined by the design. This gives an output based on whether or not the threshold is broken, but could be adjusted to give an output based on multiple thresholds to weight multiple threat levels.

These two detection systems come together to determine drone identification. If the first (WiFi) system is positive, an analog output is sent to a vibrating motor in the prototype. If the second (audio) system backs up a positive result, the output vibration is magnified. For demonstration purposes, the detection systems were running on one board while the analog output was on a second board. The two boards communicate via ESP-NOW. Each of these systems are described in added detail in the following sections.

3.2 System Block diagram


SSODD BASE STATION HARDWARE BLOCK DIAGRAM
RF SENSOR ARRAY
MICROPHONE ARRAY
ESP32
ANALOG SENSORY STIMULUS
DATA TRANSMIT TO OTHER UNITS
SSODD SATELLITE UNIT HARDWARE BLOCK DIAGRAM
BASE STATION DATA RECEIVE
ESP32
ANALOG SENSORY STIMULUS
Figure 1: Hardware Block Diagram

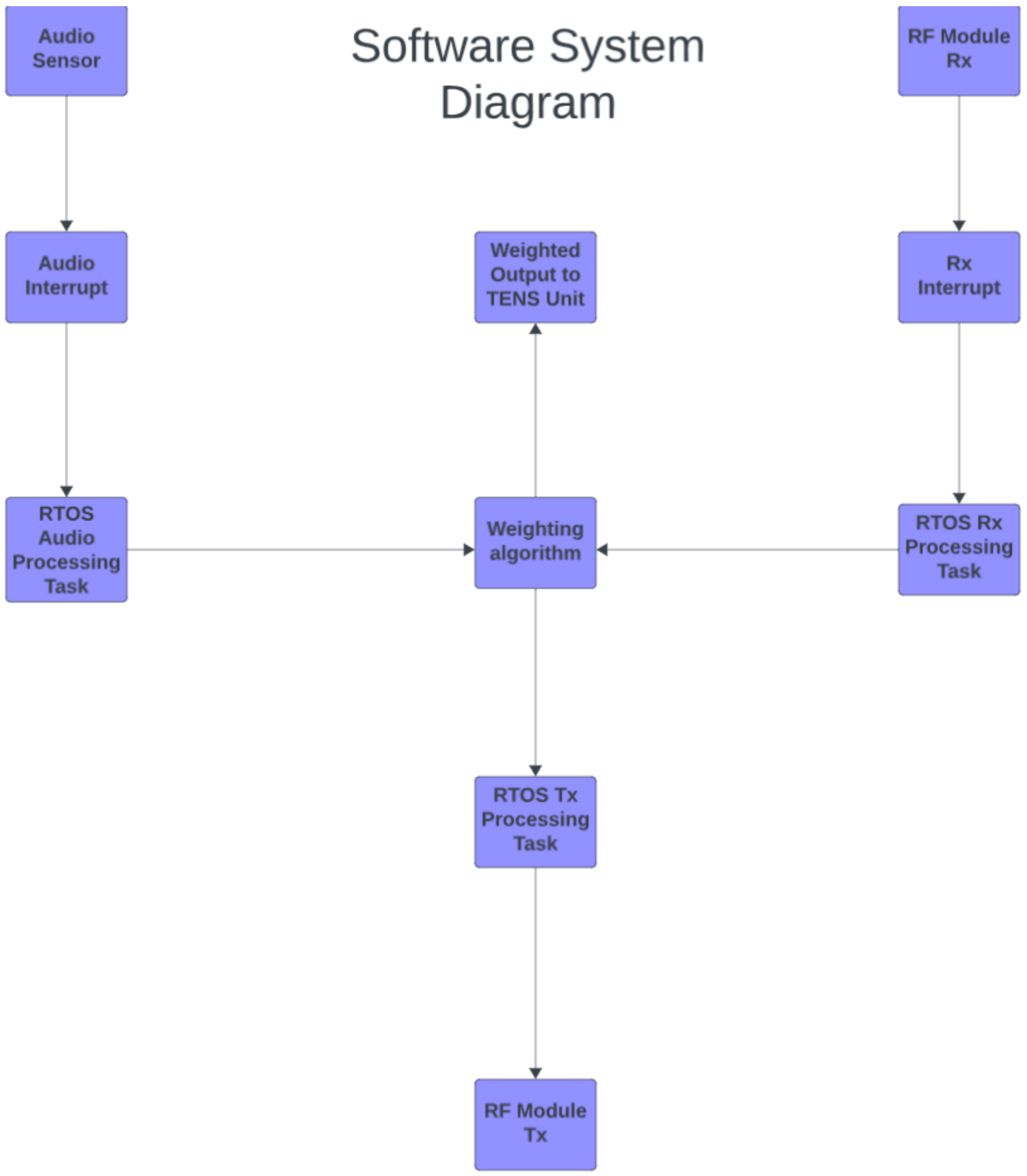


Figure 2: Overall Software Diagram

3.3 Detailed design/operation of subsystem 1

Subsystem 1: Wifi Snooping

The Wifi Snooping subsystem is characterized by two main parts: reading in mac addresses and inferencing. From a high level, the system sits at a state of standby with the ESP32 set in promiscuous mode. For a set period of time, defined as WIFI_CHANNEL_SWITCH_INTERVAL, the ESP receives a packet in the current WiFI channel. It then moves to the next channel and performs the same snooping operation. The ESP repeats this sweep through 13 Wifi channels indefinitely.

To initialize Wifi snooping, we first must set the ESP32 to the correct modes and initialize its operation properly. This is mostly performed within the wifi_sniffer_init() function, as found beginning in line . This function first initializes the system components required to utilize Wifi, such as non-volatile storage (nvs_flash_init()) to store Wifi configurations, TCP/IP networking (tcpip_adapter_init()), and the ESP32 event handler such that it can handle Wifi operations such as connections and disconnection (esp_event_loop_init()). It then initializes the ESP32's Wifi snooping capabilities as follows; we set our config 'cfg' to the default wifi configuration (WIFI_INIT_CONFIG_DEFAULT()) and passes this object into the the esp_wifi_init() function, sets the wifi country to the United States, tells the wifi driver where it is allowed to store data (esp_wifi_set_storage), starts the wifi driver code (esp_wifi_start), sets the wifi mode to promiscuous (esp_wifi_set_promiscuous(true)), and assigns a callback function to each packet received in promiscuous mode (esp_wifi_set_promiscuous_rx_cb()). This callback function handles the processing and inference related to our system. It is also important to note that the sniffer initialization function sets the max transmit power. This setting is not directly related to the wifi snooping operations and will be described in the ESPNOW communication section. The imported libraries that allow the previously functionalities to be defined are esp_wifi.h, esp_wifi_types.h, esp_system.h, esp_event.h, esp_event_loop.h, and nvs_flash.h.

Several other functions must also be defined to support the Wifi snooping operation. We assign an event handler that allows us to catch errors based on whether or not the ESP has properly handled each Wifi snooping action. We assign a function that sets the channel of our system snoops in, which is then updated in each loop. We define two other functions; wifi_sniffer_packet_type2str() converts the wifi packet type to a string, which allows us to parse the type of packet we are interested in snooping, and our callback function wifi_sniffer_packet_handler().

In order to understand how our packet handler function parses the received data for its mac address and allows us to perform our snooping algorithm, we must first define the structs used to handle the incoming wifi packet. We first define a struct called wifi_ieee80211_mac_hdr_t, which stores the intended receiver's address the sender's address, and the filtering address, along with three other fields that store information about the packet frame, duration, and ordering of sent data. Secondly, we define the struct wifi_ieee80211_packet_t to store our previously defined struct and the data payload, which we discard.

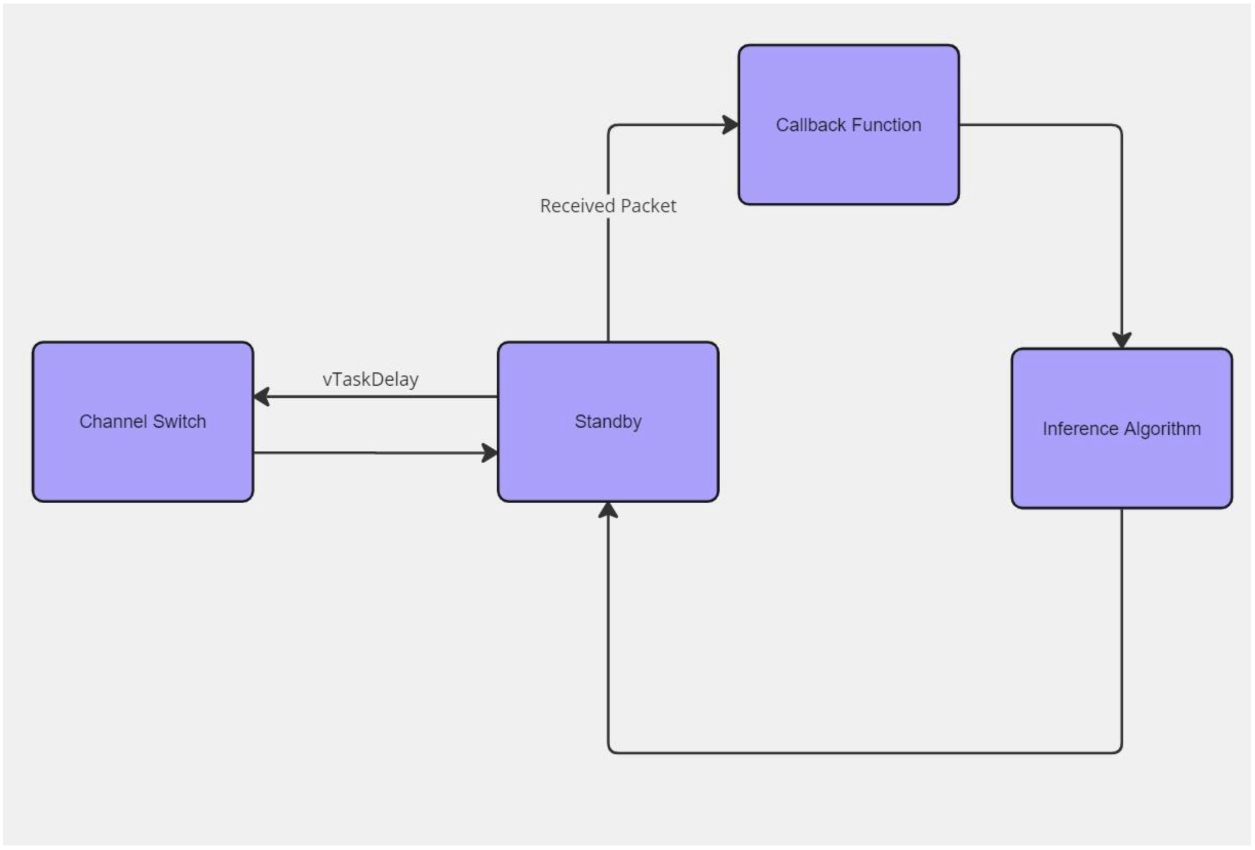


Figure 3: WiFi Snooping Flowchart

The second part of the Wifi Snooping Subsystem is the inference algorithm. This algorithm was developed with the intention of employing the first three bytes of the Mac Address, the Unique Organizational Identifier. The algorithm works in tandem with the Wifi snooping because the only data it requires is the first three bytes of each mac address. The structure of a mac address can be found below in Figure 4.

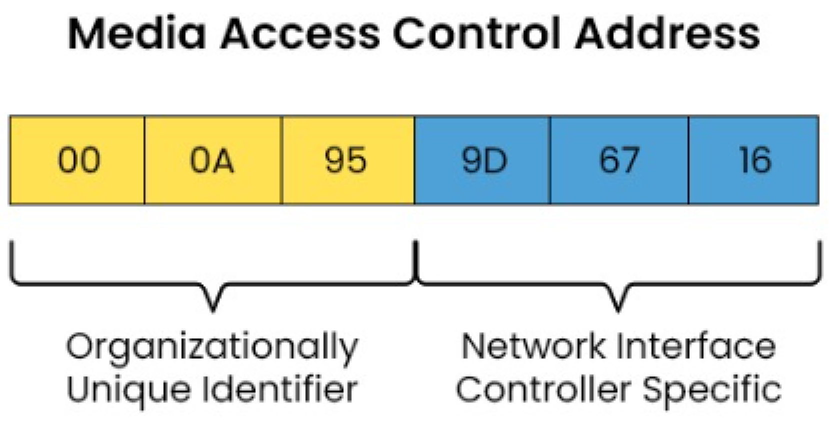
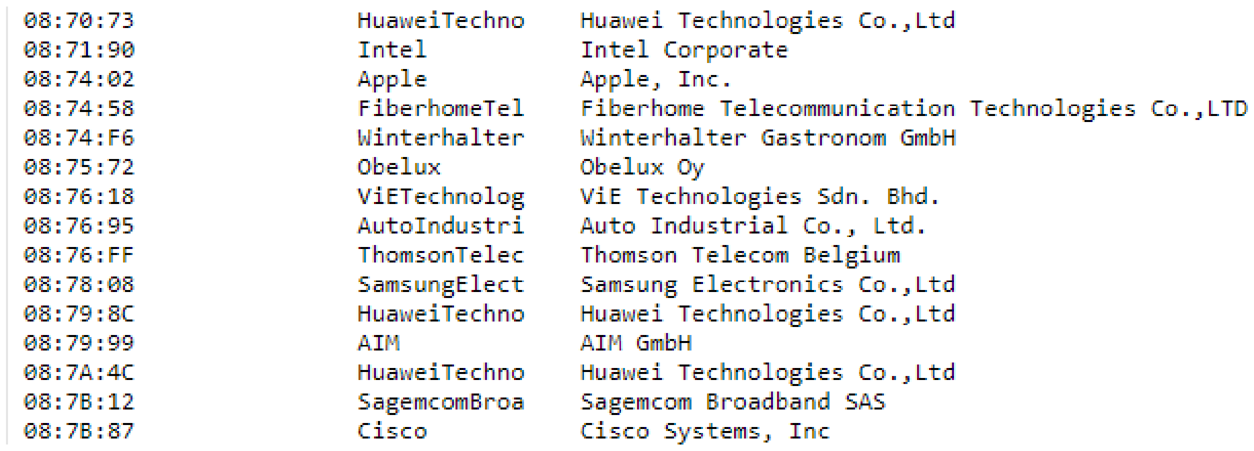

Media Access Control Address
00
0A
95
9D
67
16
Organizationally Unique Identifier
Network Interface Controller Specific

Figure 4: MAC Address Structure

By leveraging OUI's, we are able to reliably determine which mac addresses are known not to be a drone. Upon our initial testing of an off the shelf (OTS) drone, we found that the organization who designed the chip was a company called Buffalo Inc. We quickly recognized that developing a database of UOIs that are employed in drones would be a significant challenge, and would fare poorly on a battlefield. For example, if the UOI is not registered with Wireshark's Manufacturer Database, an algorithm that simply searches for known chips in drones would totally fail and provide no threat warning to those dependent on the system.



```
08:70:73    HuaweiTechno    Huawei Technologies Co.,Ltd
08:71:90    Intel           Intel Corporate
08:74:02    Apple           Apple, Inc.
08:74:58    FiberhomeTel    Fiberhome Telecommunication Technologies Co.,LTD
08:74:F6    Winterhalter    Winterhalter Gastronom GmbH
08:75:72    Obelux          Obelux Oy
08:76:18    ViETechnolog    ViE Technologies Sdn. Bhd.
08:76:95    AutoIndustri    Auto Industrial Co., Ltd.
08:76:FF    ThomsonTelec    Thomson Telecom Belgium
08:78:08    SamsungElect    Samsung Electronics Co.,Ltd
08:79:8C    HuaweiTechno    Huawei Technologies Co.,Ltd
08:79:99    AIM             AIM GmbH
08:7A:4C    HuaweiTechno    Huawei Technologies Co.,Ltd
08:7B:12    SagemcomBroa    Sagemcom Broadband SAS
08:7B:87    Cisco           Cisco Systems, Inc
```

Figure 5: Lookup Table

As a result, we elected to design our system deductively. We reasoned that it would be more favorable if the system alerted operators as a false alarm due to an unknown device within range than if the system was unable to recognize a drone's UOI. Thus, we designed the system to parse incoming UOI's based on known devices, such as Apple, Microsoft, Intel, Cisco, etc., and make an inference based on this parsing.

The algorithm performing inference is displayed in Figure 6. The algorithm begins by receiving a mac address from the Wifi Snooper and verifying whether or not its most significant byte is divisible by 4. The operation is intended to filter out mac addresses that have been randomized for encryption methods. UOIs are assigned such that the most significant byte is a multiple of 4, such as 00, 04, 08, 0C, 10, etc.. We define the filtering of randomized UOIs as a parameter of our project, since de-encrypting and inferencing based on encrypted addresses could be a whole project on its own. Once this filtering is performed, we compare the UOI to a lookup table of UOIs that was built from the Wireshark Manufacturers Databases shown in Figure 5.

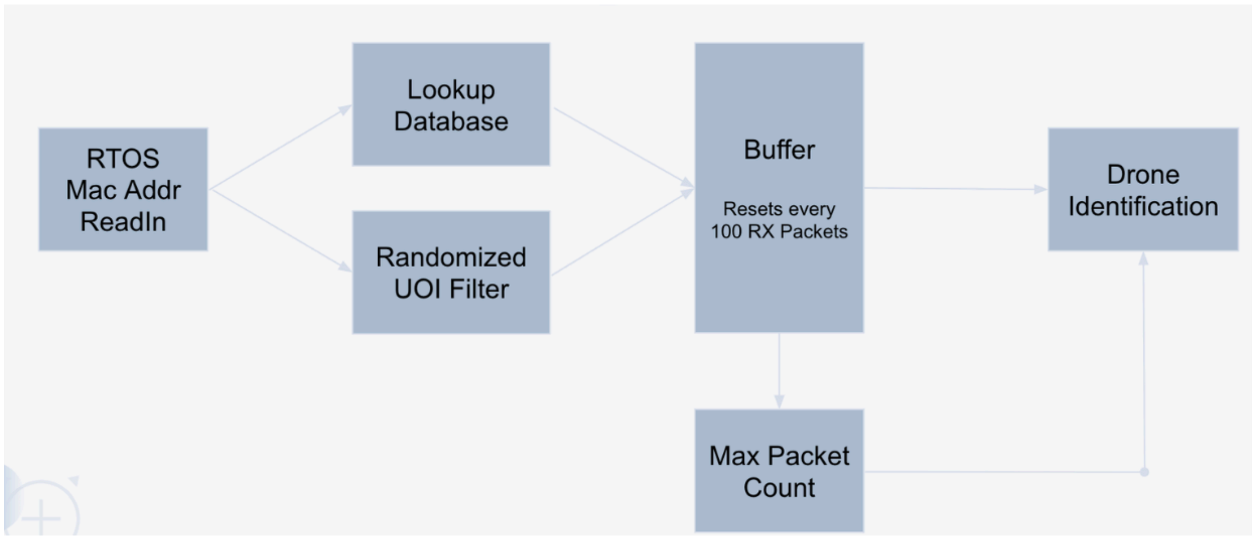


Figure 6: WiFi Snooping Algorithm

The lookup database was designed to filter out known mac addresses. Given that this project was designed and tested in the buildings around Notre Dame's campus, the existing lookup table is biased toward devices that could be found on a college campus. For example, the existing table contains UOIs from Apple, Microsoft, Cisco, Intel, Espressif, Hewlett Packard, and a range of other devices picked up by the Wifi snooper during testing.

If the mac address that was read in makes it past randomized UOI filtering and the lookup database, it is passed into a buffer. This buffer is initialized to ten devices long, which is an optimization we found during testing. With a well-designed lookup table, much of the network traffic is filtered out in the second stage of the algorithm. Therefore, we can improve our compute performance by keeping the buffer short, which will come into fruition in the final stage of the algorithm. If a UOI is added to the buffer, we assign it a count so that each time the same UOI is processed, we can add to this count and determine how many packets have been accumulated from that particular device.

After the Wifi snooper has received 150 packets, the algorithm loops through the buffer and computes the maximum number of packets received for the UOIs existing in the buffer. This number is then compared to a threshold of 5 packets, which determines whether or not the system will declare the device as a drone. If the maximum is higher than the threshold, the audio spectrum analyzer takes in its required data and determines a threat level, which is then passed to ESPNOW transmitting routine. If the maximum is not above the threshold, we pass a message that the drone has not been found and the ESPNOW routine is called. Finally, the buffer is cleared and the process restarts.

The definition of 150 packets as the length of one snooping session was defined primarily by two parameters: latency and accuracy. We intended to keep our inference latency as low as possible given the necessity for a quick notification and the error rate of ESPNOW. For accuracy, we found that a session of 150 packets and a threshold of 5 packets performed best in testing due to its ability to allow enough drone packets to surpass the threshold without allowing the number of mac addresses that sneak pass filtering to accumulate in the buffer. The session length could also have been determined by the channel switching time or by allowing some number of full channel sweeps before inferencing, but we figured that parametrizing the algorithm based on the number of received packets will allow us to make more reliable inference in both congested and non-congested environments.

Subsystem 2: ESPNOW Communication

The ESPNOW subsystem consists of a single sender and single receiver. The subsystem is designed with the intent that one sender would send to several receiving devices, acting as a functional system for a team of operators. The system was tested with several receivers, but the latency of a reliable ESPNOW communication multiple receivers at a time (or even in succession) led us to the decision to keep our final working product as a single sender and a single receiver.

The ESPNOW communication subsystem includes the esp_now.h and Wifi.h libraries. Similar to the wifi snooping, the ESPNOW communication needs to be initialized properly before any packets are sent. The first step of this initialization is to define a struct that will hold the information sent and received for each packet. This struct must be defined the same in both the sender code and the receiver code. In the sender code, we define a callback function for sending data, which will be defined later. Similarly, the receiver requires a callback function for receiving data.

To initialize the ESP32 to send data using ESPNOW, we must first declare the Wifi mode to be “Station” and call the initialization function esp_now_init(). We then must register the callback function as a callback function, and add the address we intend to broadcast to into our peer list. To initialize the ESP32 to send data using ESPNOW, we follow a similar process. We initialize ESPNOW with the same function, and we register our callback function for receiving data.

As the project progressed, we realized that the ESPNOW communication was very finicky and if a packet was sent only a single time, the likelihood that it was received was low. This realization determined the design of our communication system. Once the wifi snooping algorithm was complete, a packet is sent to the receiver address expressing whether or not a drone had been identified. When this packet is sent, the callback function is called. We designed the callback function to check the status of the sent packet for an acknowledgement from the receiver of a successful delivery. If the status defines a failed delivery, we delay the system 500ms using vTaskDelay and then resend the packet. We continue this process either until the sender is acknowledged of a successful delivery or until we try resending ten times. After ten attempts, we declare the communication a failure and the system returns to standby

The receiver board waits in a constant state of standby listening for an ESPNOW packet. When it receives a packet, it decodes the packet's information and interfaces with our analog stimulus subsystem by assigning the information stored in the packet as our analogwrite value.

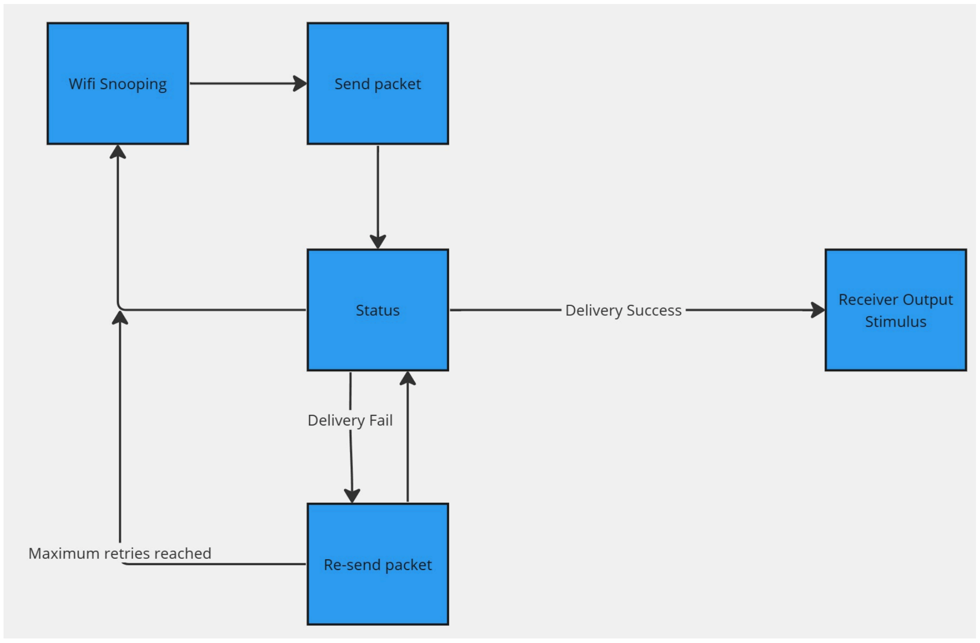


Figure 7: ESP-NOW Flowchart

The ESPNOW communication was tested in various environments and appear to function most effectively when both boards were facing toward each other and location away from nearby routers. When we initialize ESPNOW, we set it to wifi channel zero and do not change the channel during operation. We suspect that since ESPNOW is not performing dynamic channel switching like other Wifi devices, its signals are being clobbered by other devices in the area. This appears most obvious when operating near a router, as the communication between two boards fails all ten sending attempts very frequently. To solve this problem, we initialize the wifi transmit power setting to its highest setting, but we were unable to find a robust implementation of ESPNOW where most packets were received in the first sending attempt.

3.4 Subsystem 3: Audio Spectrum Analyzer Sensor

The second major sensing subsystem is the audio spectrum analyzer. The concept of the subsystem is to take in analog audio data, perform an FFT on it to see the frequency spectrum of the audio, and look at a specific range of frequencies that we know to be generated by the spinning of quadcopter propellers (roughly 200 - 500 Hz). If the power of the frequencies in that range is high enough, the system interprets this as a drone being nearby and, if a wireless signal is also detected, sets the detected threat level to 5 (the highest level). A block diagram of this subsystem can be seen below.

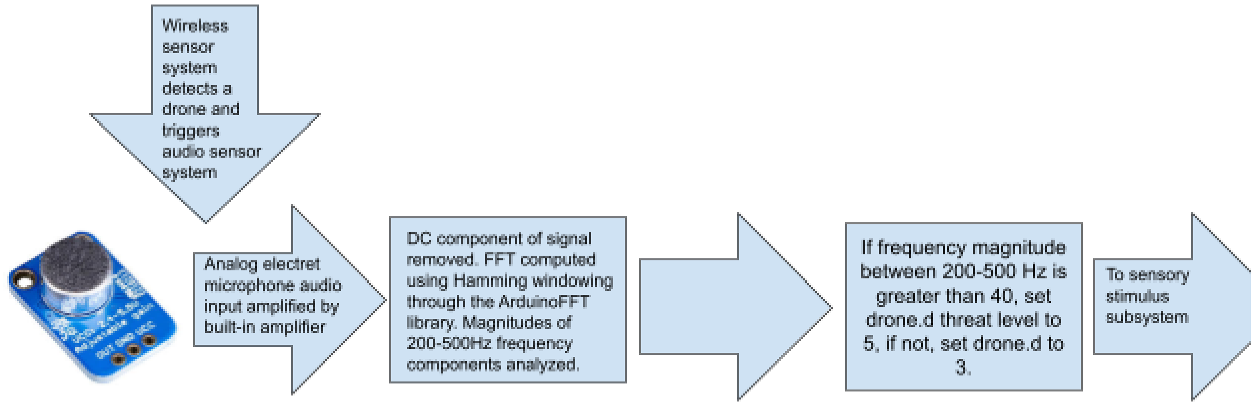
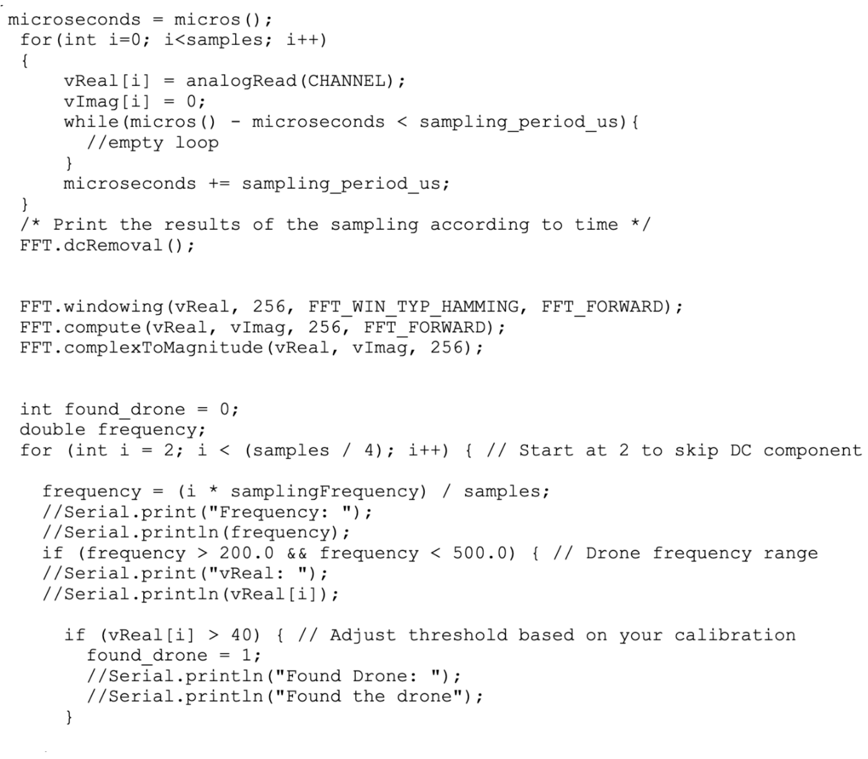

Wireless sensor system detects a drone and triggers audio sensor system
Analog electret microphone audio input amplified by built-in amplifier
DC component of signal removed. FFT computed using Hamming windowing through the ArduinoFFT library. Magnitudes of 200-500Hz frequency components analyzed.
If frequency magnitude between 200-500 Hz is greater than 40, set drone.d threat level to 5, if not, set drone.d to 3.
To sensory stimulus subsystem

Figure 8: Audio Subsystem Flowchart

Once analog data from the microphone has been sampled using an analogRead command on pin 4 collecting 256 samples at a rate of 8000 samples/sec, an instance of an ArduinoFFT object is created and is used to remove the DC offset from the data. Next an FFT is performed on the data using Hamming windowing and finally the complex results of the FFT are converted to magnitudes using the ArduinoFFT library. The resulting vector contains the magnitudes for a wide range of frequencies. This vector is searched through to find the magnitudes of the frequencies in the 200-500 Hz range and the magnitude of each of these frequencies is checked using a threshold of 40. If one of the frequencies in this range crosses the threshold, it is defined as having detected a drone, and the drone.d threat level is set to 5. This code can be seen below.

While this subsystem does work, it could be drastically improved in later iterations. The major problem with the system is that it is incredibly easy to trigger a false positive due to



```
microseconds = micros();
 for(int i=0; i<samples; i++)
 {
     vReal[i] = analogRead(CHANNEL);
     vImag[i] = 0;
     while(micros() - microseconds < sampling_period_us){
       //empty loop
     }
     microseconds += sampling_period_us;
 }
 /* Print the results of the sampling according to time */
 FFT.dcRemoval();


 FFT.windowing(vReal, 256, FFT_WIN_TYP_HAMMING, FFT_FORWARD);
 FFT.compute(vReal, vImag, 256, FFT_FORWARD);
 FFT.complexToMagnitude(vReal, vImag, 256);


 int found_drone = 0;
 double frequency;
 for (int i = 2; i < (samples / 4); i++) { // Start at 2 to skip DC component

   frequency = (i * samplingFrequency) / samples;
   //Serial.print("Frequency: ");
   //Serial.println(frequency);
   if (frequency > 200.0 && frequency < 500.0) { // Drone frequency range
   //Serial.print("vReal: ");
   //Serial.println(vReal[i]);

     if (vReal[i] > 40) { // Adjust threshold based on your calibration
       found_drone = 1;
       //Serial.println("Found Drone: ");
       //Serial.println("Found the drone");
     }
```

background noise. This could possibly be fixed by using higher quality microphones or dishes to collect sound, but it could also likely be improved by changing the threshold that the magnitude of the frequencies must reach as well as perhaps requiring a grouping of frequency components to cross the threshold as opposed to just one, which could easily be an anomaly. One of the suggestions made to us during demo day was the use of a ROC (Receiver Operating Characteristic) curve, which would allow us to plot the number of false positives vs false negatives we are getting with various threshold settings to dial in the best possible measurement criteria.

3.5 Subsystem 4: Analog Sensory Output

The analog sensory output subsystem's purpose is to alert the device's user of an incoming drone. The subsystem original plan for the subsystem was to use a TENS (Transcutaneous Electrical Nerve Stimulation) unit to deliver a moderate electrical current through the skin of the back, creating a tingling sensation proportional in intensity to the intensity of the threat perceived by the system. After initial tests we realized that the TENS unit would be large and clunky, and one of our goals was to create a device that would be as small and unobtrusive as possible. We also had safety concerns about designing our own TENS unit as the process involves higher voltages and currents that could harm a user if it malfunctioned.

Our second design for the sensory output was a simple vibration motor that could be attached to a glove or to somewhere on the body. The intensity of the motor's vibration would immediately signal to the user the likelihood of an incoming drone attack. The vibrating motors we purchased off of Amazon were quite small (10mm x 3mm) and operate at a voltage range of 2.5 - 3.7 V DC with a rated speed of 12,000 RPM. The ground wire of the motor was connected to the ground pin of the board through connector J4. The signal wire of the motor was connected to analog pin 5 of the ESP32 through J4. A flow chart for the operation of the subsystem can be seen below.

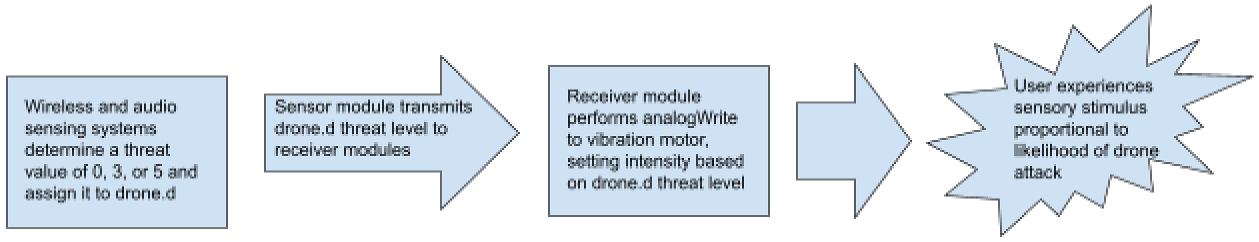

Wireless and audio sensing systems determine a threat value of 0, 3, or 5 and assign it to drone.d
Sensor module transmits drone.d threat level to receiver modules
Receiver module performs analogWrite to vibration motor, setting intensity based on drone.d threat level
User experiences sensory stimulus proportional to likelihood of drone attack

Figure 9: Analog Sensory Flowchart

The code for operating the sensory output can be split into two segments: the code for the unit that senses drones and broadcasts a threat level, and the code for all receiver units that turn the threat level into a sensory stimulus. The broadcaster unit code can be seen below.

```
if (max >=5){
  //Serial.println("Found the drone at UOI of %02x:%02x:%02x\n", lookup[max_index][0], lookup[max_index][1], lookup[max_index][2]);
Serial.print("Found the drone at UOI of ");
Serial.print(lookup[max_index][0], HEX); // Print first byte as hexadecimal
Serial.print(":");
Serial.print(lookup[max_index][1], HEX); // Print second byte as hexadecimal
Serial.print(":");
Serial.println(lookup[max_index][2], HEX);
strcpy(drone.a, "Found the drone at UOI of ");
drone.b = lookup[max_index][0];
drone.c = lookup[max_index][1];
if (found_drone == 1){
  drone.d = 5;
}
else{
  drone.d = 3;
}


sendData(broadcastAddress2, drone);
//vTaskDelay(pdMS_TO_TICKS(RETRY_DELAY_MS));
//sendData(broadcastAddress2, drone);
// esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &BME280Readings, sizeof(BME280Readings));
//   // Send message via ESP-NOW
//   while (ESP_NOW_SEND_SUCCESS !=0){
//     esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &BME280Readings, sizeof(BME280Readings));
//   }
}
else{
strcpy(drone.a, "Did not find a drone ");
drone.b = 0;
drone.c = 0;
drone.d = 0;
found_drone = 0;
//xQueueSend(dataQueue, &drone, portMAX_DELAY);
sendData(broadcastAddress2, drone);
//vTaskDelay(pdMS_TO_TICKS(RETRY_DELAY_MS));
//sendData(broadcastAddress3, drone);
}
```

Figure 10: Analog Output Code

Essentially, after searching for both wireless and auditory signs of drones, one of three values is assigned to the threat level object drone.d. If a wireless signal for a drone is found but there is no auditory sign of drone presence, drone.d is set to 3. If both wireless and auditory signals are sensed, drone.d is set to 5. If neither signal is sensed, drone.d is set to 0. There is currently no code for if only an auditory signal is sensed because the auditory signal is currently much more prone to false positives due to poor quality microphones, so an auditory signal is currently only being searched for if a wireless signal has already been found. In future iterations of this project with more sensitive hardware the code could be changed to also set drone.d to 3 if only an audio signal is detected. Once a value of drone.d has been set, it is broadcasted to all other units in the area, and the value is processed by the receiver code below.

```
void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len) {
  Serial.println("Data received.");
  memcpy(&myData, incomingData, sizeof(myData));
  Serial.print("Bytes received: ");
  Serial.println(len);
  Serial.print("Message: ");
  Serial.println(myData.a);
  Serial.print(myData.b, HEX);
  Serial.print(":");
  Serial.print(myData.c, HEX);
  Serial.print(":");
  Serial.println(myData.d, HEX);
  if (myData.d > 0){
    analogWrite(ALERT, 51*myData.d);
  }
  else{
    analogWrite(ALERT, myData.d);
  }
}
```

Figure 11: Broadcast Code

The receiver code operates by taking the value of drone.d and performing an analogWrite command to the pin designated as ALERT (we use pin 5) to turn the motor on to a specific intensity based on the received threat level. If drone.d is 3 or 5, the analogWrite command sets the motor to a level of 153 or 255 respectively. If drone.d is 0, the analogWrite command turns off the motor by setting it to zero.

4 System Integration Testing

4.1 Describe how the integrated set of subsystems was tested.

To integrate our subsystems we started by testing each subsystem separately. For the wireless subsystem we checked to make sure that turning on the drone nearby would display a message in the serial port stating that a drone was found. For the audio subsystem we used an LED that would turn on when the frequency of the drones propellers was sensed and tested it with audio recordings of those frequencies. To test the ESPNow communication between chips we checked to make sure that a message could be sent from one ESP32 development board to the other.

The next step was to design a board schematic and layout that would include all the necessary pieces (ESP32, CP2012 for USB programming, pins for connecting the vibrating motor and the microphone, USB-C and battery ports, etc) for both the sensing and receiving board configurations. When the completed boards arrived we built them and tested to see whether the boards were programmable. An issue was discovered in uploading our code to the boards, and after comparing with the schematic and testing for continuity between pieces, it was discovered that the ground connections underneath the ESP32 were not making a solid connection, but that mild pressure applied to the chip would make the connection, so the issue was solved by using binder clips to hold down the chip. Once the boards were programmable we made two configurations of the board, one with a microphone for sensing and one with a motor for providing stimulus based on received data from the sensing unit. After our hardware was properly configured, it was time to begin code integration.

To begin our code integration we started by integrating the code for the wireless sensor and the audio spectrum analyzer together to create a program that would continuously search for wireless signals, and if a drone's signal was found, search for audio signals. We decided not to continuously search for both wireless and audio signals at the same time for two reasons: first, to save processing power on the ESP32, and second because the audio sensor was so sensitive that many of its readings were false positives if the room got too noisy. The combination of these two subsystems would output a threat level of either 0 (no drone), 3 (wifi signal detected but no audio) or 5 (wifi and audio detected). Once we found that the integrated subsystems were properly sensing a drone we integrated the ESPNow communication between boards and verified that the receiver board was properly receiving the threat level from the sensing board. The final step of integration was installing the vibration motor on the receiver board and integrating the code on the receiver board to change the motor's vibration based on received threat level. When this was all working properly we were ready for demo day.

4.2 Show how the testing demonstrates that the overall system meets the design requirements

In our final testing/demo of the project we tied down a drone to a nearby handrail and set up a sensing unit with a microphone and a receiving unit with a vibrating motor. When the drone was off the motor did not buzz, showing that no drones were within the range of the device.

One or two times during the demo day strange devices that were not on our MAC address lookup list entered the range and triggered a low threat level, but we were able to update our list as this happened. This was an expected difficulty, and a part of our setup of the device involves excluding the MAC addresses of the types of devices you expect to encounter in your specific environment as devices that appear to be "atypical" will change based on where the device is used.

When the drone was turned on it was automatically detected by the sensor unit and the threat level was sent to the receiver unit, resulting in a low intensity vibration signaling a possible drone in the location. The only issue that we encountered in the final testing of this part of the system was that occasionally ESPNow would fail to send the packet containing the updated threat level from the sensing unit to the receiving unit, resulting in a latency in the change of the vibration based on threat level. In a future iteration of the project we would fix this with a better antenna.

Once a drone had been detected via WiFi, the audio spectrum analyzer system would start searching for the sound of drone propellers. We would rev up the drone's propellers since it was tied down and wouldn't move, and the sound of the rotors would activate the higher threat level resulting in a more intense vibration. This was the area where our final test struggled the most on demo day as the high level of ambient noise in the room from all the visitors and other teams would frequently trigger the high threat level as other groups would end up making a sound in the frequency range momentarily, triggering the system, and then the high latency between the sensor and the receiver would mean that the motor would continue to vibrate at a high threat level for some time even though there was no propeller movement. This could be fixed in a future iteration by raising the threshold for triggering the system and requiring a grouping of frequencies to meet the threshold instead of only one. Even with issues of our sensitivity being too high, the unit always successfully picked up the sound of the drone's propellers and translated it to a high threat level that would result in the motor vibrating at its most intense level.

We believe that this demo showed that we had met our design requirements by accurately detecting a drone in a location using a two-factor sensing method (wireless and audio) and producing some sort of sensory feedback (vibration motor) that could be interpreted as various levels of threat from a drone attack.

5 Users Manual

5.1 How to install your product

Our boards are designed with the intention of being modular, so each board is capable of acting as the main sensor or as a satellite receiver. The product would likely be installed on the equipment of a team of operators, with each individual operator possessing their own board. From our initial vision, the sensor board would be attached to the helmet of the designated "drone" operator, i.e. the team member who is responsible for administering threats electronically and notifying other team members. Our current analog stimulus is intended to be placed on the non-dominant arm of each operator, assuming that the dominant hand is required for tasks that are best not interrupted. With this setup, the board should be placed in the front pack of each operator such that it is not subject to significant force or motion. In an iteration of our product that is ready for use, we intend to place a protective covering around the board to mitigate any forceful impacts.

5.2 How to setup your product

For the WiFi detection to work, it must be "trained" for its location of use. Expected MAC addresses must be added to the lookup table, and it is recommended to test the device multiple times at various locations to detect any unexpected, friendly, MAC addresses. This is inherently different for each location, and in our demonstration case it is set for the environment at Notre Dame. It is expected that more complex or variable unknowns could be present in the environment of the intended use case.

5.3 How the user can tell if the product is working

It is clear that the product is working if the analog output is activated upon detection of an unrecognized MAC address. This confirms the working state of the detection, communication, and output protocol. Additionally, if a tone in the 200-500 Hz range is played and the output is amplified, users can confirm the operation of the audio analysis.

5.4 How the user can troubleshoot the product

To troubleshoot, users must identify which subsystems are malfunctioning. For the prototype, much of the troubleshooting must be done via a serial monitor. The WiFi detection system outputs the MAC addresses of any unrecognized signal when detected. The audio system can also output frequency ranges above the defined threshold as they are detected to the serial monitor. If either of these are not working properly, there is a clear problem with the detection systems. Next, it is necessary to troubleshoot the output. In the prototype, one can simply send a high analog signal to the vibrating motor to make sure it is working properly. If all of these work, then the issue lies with the ESP-NOW communication between multiple boards. Common issues arise here including the communication being blocked on popular WiFi channels.

6 To-Market Design Changes

On April 25th, our team got the opportunity to present our research to end users and USSOCOM tech developers at the SOCOM Ignite closing ceremony in Boston, Massachusetts. These stakeholders' direct input gave us insight for how our design should be further developed. Chief among these was a suggestion for the device's stimulus given to us by the Command Sergeant Major of SEAL Team Six. He told us that SOF was currently developing a set of eye protection with a heads up display built into the lenses. He suggested that we move from a physical stimulus to an alert that flashes over the operator's eyes. While the challenge originally given to us was to design a wearable device, end users noted that too many wearables are currently being implemented, cluttering the forearm. Switching to a heads up display method will allow us to transfer the location of our board to anywhere on the operator's kit. Another concern was frequent false alarms. It is human nature to dismiss warnings after too many false alarms. To counter this, we can develop a threshold in the code that does not alert the operator unless the threat level is high.

We would like to improve our minimum viable product by increasing the quality of our sensors. Rather than our current microphone, we would like to use four long distance directional microphones. The greater distance that we can hear the propellers, the more time the operators have to react. We would also like to employ stronger antennas to prevent frequent failures in packet sending from one board to another.

7 Conclusions

Our team was challenged by the United States Special Operations Command to develop a wearable early warning device for drone detection. Hobby drones paired with explosives are becoming more and more present on the battlefield, and special forces do not currently have a method for detecting these threats.

Our device detects drones on the battlefield through sniffing WiFi communications between the drone and its controller and listening in for the distinct sound of drone propellers. A master board detects the signal and sound and calculates a threat likelihood level. This threat likelihood level is broadcast to all other boards in the area, and a physical stimulus is output proportional to the threat likelihood level.

In order to identify a drone from its wifi communication, we sniff nearby networks for packets. In each packet's header exists a Unique Organizational Identifier. We compare this UOI to a lookup table of common UOIs that we know are not threats. Unrecognizable UOIs are considered possible drones. When enough packets are flagged as possible drones, the threat likelihood level increases.

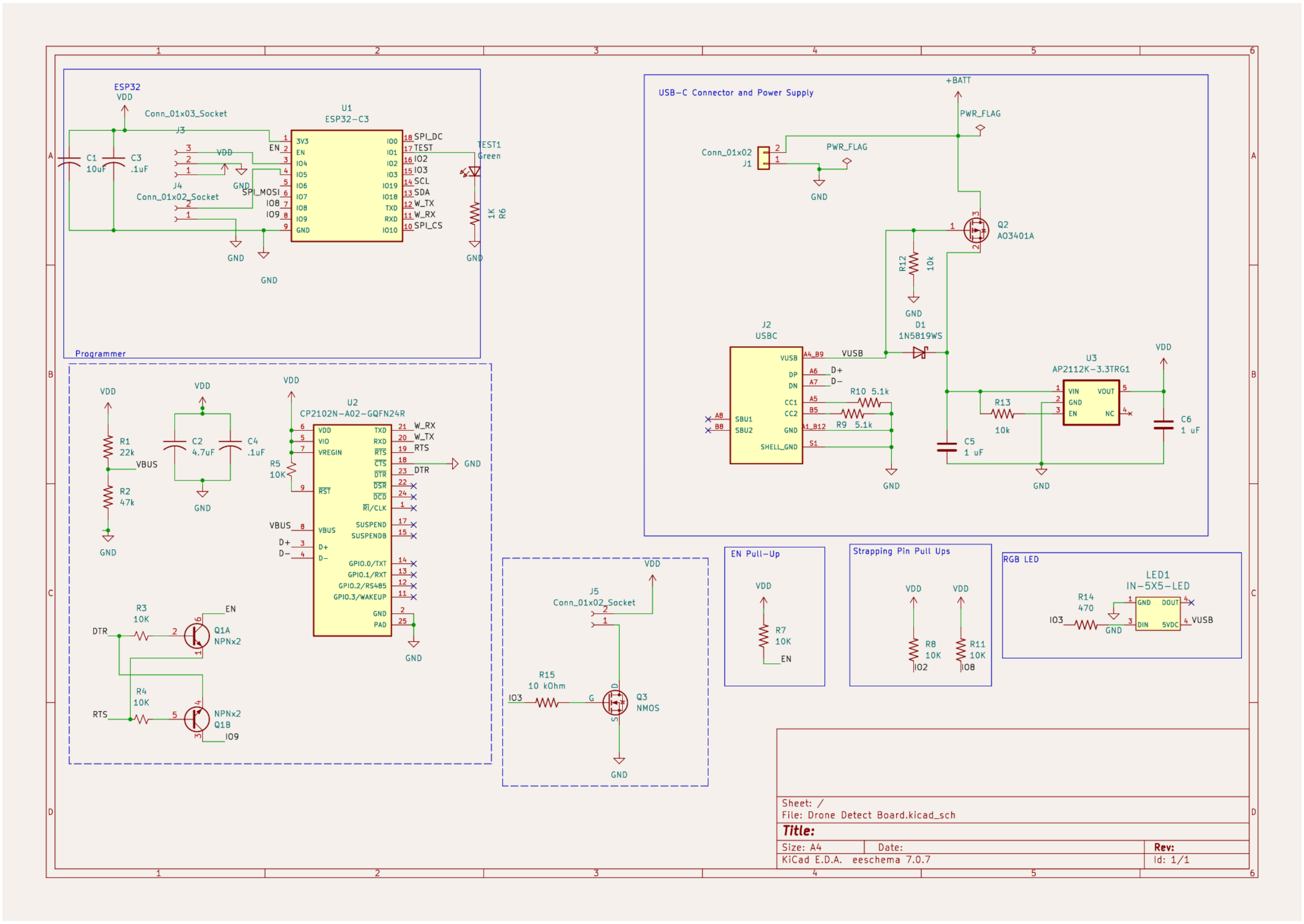
To detect a drone based on sound, we read in ambient microphone data. A Fast Fourier Transform is performed on the data. If there is a high enough spike at the key range of 200 Hz to 500 Hz, the threat likelihood increases.

We would improve our design by increasing the quality of our sensors. Particularly, improving our microphone to four long distance directional microphones in order to provide data on direction as it could add vital information for the output. A final product would also interface with the aforementioned developing heads-up-display included in the wearable eye protection. Another optimization of the design is mesh networking, allowing multiple sensors to communicate and provide localized data based on RSSI measurements from each sensor.

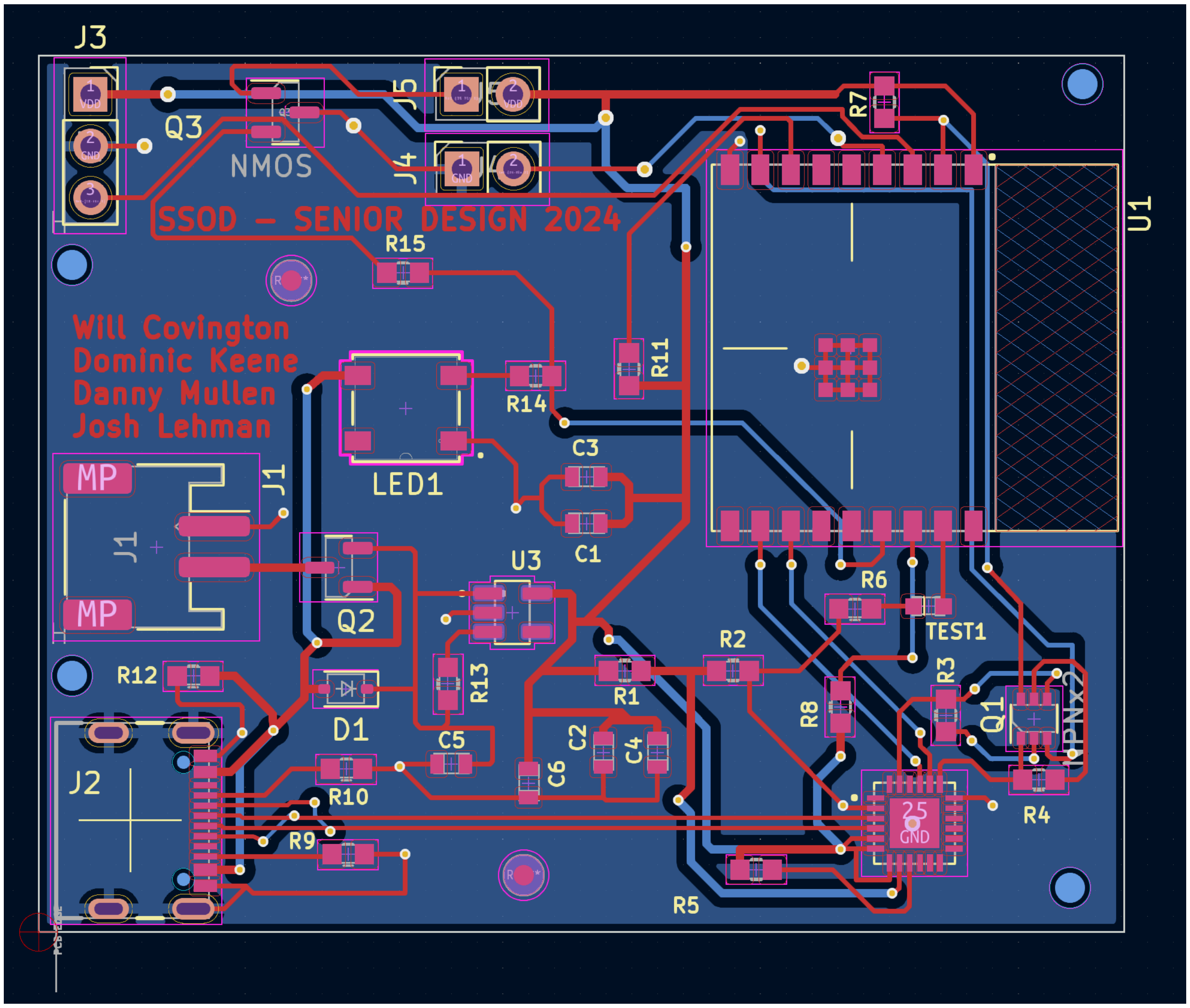
In conclusion, our project did achieve the end goals we set forth in the beginning of the process, despite some changes along the way. The robust system can successfully detect a drone, but there are several areas which can be improved in future iterations as outlined. We believe that our research and implementation could at some point provide a valuable resource for SOCOM and the future development of similar products.

8 Appendices

8.1 Hardware Schematics and Layout



SSODD Board Schematic



SSODD Board Layout: J1 connector is for battery connection if not using USB-C power, J2 connects to USB-C for power and programming, J3 connects to an amplified electret microphone for audio spectrum analysis, J4 is connected to programmable analog pin 5 for sensory stimulus output, while J5 serves as a low side switch as another possible option for sensory stimulus output.

8.2 Software Listings

WiFi Snooping/Sensor Code:

```
#include "esp_wifi.h"
#include "esp_wifi_types.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "nvs_flash.h"
#include "driver/gpio.h"
#include "stdlib.h"
#include "Arduino.h"
#include <esp_now.h>
#include <WiFi.h>
#include <Wire.h>
#include <string.h>
#include "arduinoFFT.h"

#define LED_GPIO_PIN                     5
#define WIFI_CHANNEL_SWITCH_INTERVAL  (500)
#define WIFI_CHANNEL_MAX               (13)
#define DEVICE_COUNT                   10
#define DATABASE_COUNT                  89
#define MAX_SENDS                       10
#define RETRY_DELAY_MS  500


const uint16_t samples = 256; //This value MUST ALWAYS be a power of 2
const double samplingFrequency = 8000;
unsigned int sampling_period_us;
unsigned long microseconds;

#define CHANNEL 4
#define SCL_INDEX 0x00
#define SCL_TIME 0x01
#define SCL_FREQUENCY 0x02
```

```
#define SCL_PLOT 0x03

uint8_t level = 0, channel = 1;
esp_now_peer_info_t peerInfo;
static wifi_country_t wifi_country = {.cc="CN", .schan = 1, .nchan = 13};
//Most recent esp32 library struct

int lookup[DEVICE_COUNT][4];
int next_open_slot = 0;
int total_packets = 0;
int retryAttempts = 0;
int database[DATABASE_COUNT][3] = {{132, 241, 71}, // Cisco Systems, Inc
                       {28, 95, 43}, // D-Link International
                       {60, 70, 216}, //Tp-Link Technologies Co.,Ltd.
                       {220, 133, 222}, //AzureWave Technology Inc
                       {252, 170, 129}, // Apple, Inc.
                       {56, 104, 147},//Intel Corporate
                       {184, 138, 96}, // Intel Corporate
                       {204, 71, 64}, //AzureWave Technology Inc.
                       {40, 146, 74}, // Hewlett Packard
                       {72, 137, 231}, // Intel
                       {216,58, 221}, // Raspberry Pi Trading Ltd
                       {68, 218, 48}, //Apple, Inc.
                       {208, 83, 73}, // Liteon Technology Corporation.
                        {216, 58, 221}, // Raspberry Pi Trading Ltd
                        {244, 212, 136}, // Apple, Inc.
                        {220, 133,222}, // AzureWave Technology Inc.
                        {84, 50, 199}, // Apple, Inc.
                        {52, 225, 45}, // Intel Corporate
                        {208, 63, 170}, // Apple, Inc.
                        {56, 249, 211}, // Apple, Inc.
                        {36, 246, 119}, // Apple, Inc
                        {76, 68, 91}, // Intel Corporate
                        {132, 239, 24}, //Intel Corporate
                        {88, 185, 101}, //Apple, Inc.
                        {220, 233, 148}, //Cloud Network Technology
Singapore Pte. Ltd.
                        {144, 72, 154},// Hon Hai Precision Ind. Co.,Ltd.
                        {236, 92, 104}, // Chongqing Fugui Electronics
Co.,Ltd.
```

```
                        {16, 104, 56}, // AzureWave Technology Inc.
                        {64, 236, 153}, // Intel Corporate
                        {212, 109, 109}, // Intel Corporate
                        {140, 69, 0}, // Murata Manufacturing Co., Ltd.
                        {116, 4, 241}, // Intel Corporate
                        {184, 17, 75}, // Cisco Systems, Inc
                        {144, 76, 229}, // Hewlett Packard Enterprise â€“
WW Corporate
                        {224, 208, 69}, // Intel Corporate
                        {200, 137, 243},  // Apple, Inc.
                        {40, 193, 160}, // Apple, Inc
                        {60, 33, 156}, // Intel
                        {208, 57, 87}, // Liteon Technology Corporation
                        {64, 26, 88}, //Wistron Neweb Corporation
                        {104, 122, 100}, // Intel Corporate
                        {28, 134, 130},  // Apple, Inc
                        {216, 93, 76}, // Tp-Link Technologies Co.,Ltd.
                        {0x20, 0x9C, 0xB4}, // Hewlett Packard
                        {0x2C, 0x4D, 0x54},  //ASUSTek COMPUTER INC.
                        {0xDC, 0x21, 0x5C}, // Intel Corporate
                        {0xA8, 0x8F, 0xD9}, // Apple, Inc.
                        {0x98, 0xA8, 0x65},// Intel
                        {0x90, 0xE8, 0x68},//AzureWave Technology Inc.
                        {0xD8, 0xBE, 0x1F}, // Apple Inc
                        {0x98, 0xAF, 0x65},  // Intel
                        {0xAC, 0xC9, 0x06}, // Apple
                        {0xB8, 0xA5, 0x35}, // Vantiva
                        {0x60, 0x3D, 0x26}, // Vantiva
                        {0x94, 0x6A,0x77}, // Vantiva
                        {0x70, 0xA7, 0x41}, //Ubiquiti Inc
                        {0xD8, 0x8E, 0xD4}, //eero inc.
                        {0x60, 0x22, 0x32}, //Ubiquiti Inc
                        {0x6C, 0x4B, 0xB4},// HUMAX Co., Ltd.
                        {0xB8, 0x5E, 0x71}, // VantivaUSA
                        {0xC0, 0x94, 0x35}, //ARRISGroup
                        {0xF8, 0x9E, 0x28}, // CiscoMeraki
                        {0x48, 0xA2, 0xE6}, // Resideo
                        {0xE8, 0x48, 0xB8}, //TPLink
                        {0x80, 0xCC, 0x9C},//Netgear
                        {0x2C, 0x5A, 0x0F}, //Cisco Systems, Inc
                        {0x2C, 0x76, 0x8A}, //Hewlett Packard
                        {0x74, 0x3A, 0xF4},  // Intel Corporate
                        {0xB0, 0xBE, 0x76},
                        {0x00, 0x45, 0xE2 },
                        {0x64, 0xBC, 0x58},
                        {0xC4, 0xBD, 0xE5},
                        {0xD4, 0xF9, 0x8D},
                        {0x1C, 0x71, 0x25},
                        {0xB0, 0x68, 0xE6},
                        {0xF8, 0xFF, 0xC2},
                        {0x54, 0x35, 0x30},
                        {0x64, 0xD6, 0x9A},
                        {0xC8, 0x2E, 0x18},
                        {0xE8, 0x5F, 0x02},
                        {0x08, 0xE6, 0x89},
                        {0x3C, 0xE9, 0x0E}, // ESPRESSIF
                        {0xC0, 0xB5, 0xD7}, //Chongqing Fugui Electronics
Co.,Ltd.
                        {0x00, 0x15, 0x61},
                        {0x1C, 0x57, 0xDC},
                        {0xC8, 0xB2, 0x9B},
                        {0xDC, 0xDA, 0x0C},
                        {0xC8, 0x5E,0xA9},
                        {0x3C, 0x22, 0xFB}};

typedef struct {
  unsigned frame_ctrl:16;
  unsigned duration_id:16;
  uint8_t addr1[6]; /* receiver address */
  uint8_t addr2[6]; /* sender address */
  uint8_t addr3[6]; /* filtering address */

  unsigned sequence_ctrl:16;
  uint8_t addr4[6]; /* optional */
} wifi_ieee80211_mac_hdr_t;

//uint8_t broadcastAddress[] = {0xDC, 0xDA, 0x0C, 0xFE, 0xA4, 0xE0}; //
SSOD board 1
uint8_t broadcastAddress2[] = {0xDC, 0xDA, 0x0C, 0xFD, 0xEE, 0x54}; //
SSOD board 2
```

```
//uint8_t broadcastAddress3[] = {0xDC, 0xDA, 0x0C, 0xFD, 0xEE, 0x50}; //
SSOD board 3

String success;

void addEspNowPeer(uint8_t *macAddr) {
  esp_now_peer_info_t peerInfo = {};
  memcpy(peerInfo.peer_addr, macAddr, 6);
  peerInfo.channel = 0;  // use the current channel
  peerInfo.encrypt = false;

  // Add peer
  if (esp_now_add_peer(&peerInfo) != ESP_OK) {
    Serial.println("Failed to add peer");
  }
}
//Structure example to send data
//Must match the receiver structure
typedef struct struct_message {
    char a[32];
    int b;
    int c;
    int d;
} struct_message;

double vReal[samples];
double vImag[samples];

/* Create FFT object */
ArduinoFFT<double> FFT = ArduinoFFT<double>(vReal, vImag, samples,
samplingFrequency);

// Create a struct_message called BME280Readings to hold sensor readings
struct_message drone;


typedef struct {
  wifi_ieee80211_mac_hdr_t hdr;
  uint8_t payload[0]; /* network data ended with 4 bytes csum (CRC32) */
} wifi_ieee80211_packet_t;
```

```
void sendData(const uint8_t* broadcastAddress, struct_message data) {
  // Send message via ESP-NOW
  //Serial.println("sending data...");
  esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &data,
sizeof(data));




}





void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {

  if (status != ESP_NOW_SEND_SUCCESS) {
    // If transmission failed and maximum retry attempts not reached,
retry sending
    if (retryAttempts < MAX_SENDS) {
      //Serial.println("Retrying data transmission...");


      sendData(mac_addr, drone); // Retry sending data
      vTaskDelay(pdMS_TO_TICKS(RETRY_DELAY_MS));
      retryAttempts++; // Increment retry counter
    } else {

      Serial.println("Delivery Fail");
      retryAttempts = 0; // Reset retry counter
    }
  } else {
    Serial.println("Delivery Success");
    Serial.print("Number of tries: ");
    Serial.println(retryAttempts);
    retryAttempts = 0; // Reset retry counter on successful transmission
  }
}
```

```
static esp_err_t event_handler(void *ctx, system_event_t *event);
static void wifi_sniffer_init(void);
static void wifi_sniffer_set_channel(uint8_t channel);
static const char
*wifi_sniffer_packet_type2str(wifi_promiscuous_pkt_type_t type);
static void wifi_sniffer_packet_handler(void *buff,
wifi_promiscuous_pkt_type_t type);

esp_err_t event_handler(void *ctx, system_event_t *event)
{
  return ESP_OK;
}

void wifi_sniffer_init(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_country(&wifi_country) ); /* set country
for channel range [1, 13] */
  ESP_ERROR_CHECK( esp_wifi_set_storage(WIFI_STORAGE_RAM) );
  ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_NULL) );

  ESP_ERROR_CHECK( esp_wifi_start() );
  ESP_ERROR_CHECK(esp_wifi_set_max_tx_power(78));
  esp_wifi_set_promiscuous(true);
  esp_wifi_set_promiscuous_rx_cb(&wifi_sniffer_packet_handler);
}

void wifi_sniffer_set_channel(uint8_t channel)
{
  esp_wifi_set_channel(channel, WIFI_SECOND_CHAN_NONE);
}

const char * wifi_sniffer_packet_type2str(wifi_promiscuous_pkt_type_t
type)
{
  switch(type) {
```

```
  case WIFI_PKT_MGMT: return "MGMT";
  case WIFI_PKT_DATA: return "DATA";
  default:
  case WIFI_PKT_MISC: return "MISC";
  }
}

void wifi_sniffer_packet_handler(void* buff, wifi_promiscuous_pkt_type_t
type)
{
  if (type != WIFI_PKT_MGMT)
    return;

  const wifi_promiscuous_pkt_t *ppkt = (wifi_promiscuous_pkt_t *)buff;
  const wifi_ieee80211_packet_t *ipkt = (wifi_ieee80211_packet_t
*)ppkt->payload;
  const wifi_ieee80211_mac_hdr_t *hdr = &ipkt->hdr;

  int x = hdr->addr2[0]%4;

if (x != 0 ){

  goto end;
}

for (int i = 0; i < DATABASE_COUNT; i++){
  int count= 0;
  for (int j = 0; j< 3; j++){
    if (hdr->addr2[j] == database[i][j]){
      count++;
      if (count == 3){
        goto end;
      }
    }
  }
}



for (int i = 0; i < next_open_slot+1; i++){
  int verified = 0;
  int count= 0;
  for (int j = 0; j< 3; j++){


     if (lookup[i][j] == hdr->addr2[j]){
      verified++;
      if (verified == 3){
        //printf("found at position: %d\n",i);

        lookup[i][3]+=1;
        goto end;
      }
     }
  }

  if (i == next_open_slot ){
      for (int j = 0; j< 3; j++){
      lookup[i][j] = hdr->addr2[j];

  }
  lookup[i][3] =1;


   next_open_slot++;
   goto end;



  }


}


end:
total_packets++;
int max = 0;
int max_index = 0;
//printf("Number of Packets: %d\n", total_packets);

  if (total_packets > 150){
  // prints out the matrix
    for (int i = 0; i < DEVICE_COUNT; i++){
      for (int j = 0; j < 4; j++){

        printf("%02x:", lookup[i][j]);

  }
printf("\n");
  if (lookup[i][3] > max){
      max = lookup[i][3];
      max_index = i;
  }

}
 microseconds = micros();
  for(int i=0; i<samples; i++)
  {
      vReal[i] = analogRead(CHANNEL);
      vImag[i] = 0;
      while(micros() - microseconds < sampling_period_us){
        //empty loop
      }
      microseconds += sampling_period_us;
  }
  /* Print the results of the sampling according to time */
  FFT.dcRemoval();


  FFT.windowing(vReal, 256, FFT_WIN_TYP_HAMMING, FFT_FORWARD);
  FFT.compute(vReal, vImag, 256, FFT_FORWARD);
  FFT.complexToMagnitude(vReal, vImag, 256);


  int found_drone = 0;
  double frequency;
  for (int i = 2; i < (samples / 4); i++) { // Start at 2 to skip DC
component

    frequency = (i * samplingFrequency) / samples;

    if (frequency > 200.0 && frequency < 500.0) { // Drone frequency range
```

```
      if (vReal[i] > 40) { // Adjust threshold based on your calibration
        found_drone = 1;

      }

    }
  }
if (max >=5){
  //Serial.println("Found the drone at UOI of %02x:%02x:%02x\n",
lookup[max_index][0], lookup[max_index][1], lookup[max_index][2]);
Serial.print("Found the drone at UOI of ");
Serial.print(lookup[max_index][0], HEX); // Print first byte as
hexadecimal
Serial.print(":");
Serial.print(lookup[max_index][1], HEX); // Print second byte as
hexadecimal
Serial.print(":");
Serial.println(lookup[max_index][2], HEX);
strcpy(drone.a, "Found the drone at UOI of ");
drone.b = lookup[max_index][0];
drone.c = lookup[max_index][1];
if (found_drone == 1){
  drone.d = 5;
}
else{
  drone.d = 3;
}



sendData(broadcastAddress2, drone);

}
else{
strcpy(drone.a, "Did not find a drone ");
drone.b = 0;
drone.c = 0;
drone.d = 0;
found_drone = 0;
```

```
sendData(broadcastAddress2, drone);

}

total_packets = 0;
memset(lookup, 0, sizeof(lookup));
next_open_slot = 0;

  }

}



// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 5 as an output.
  Serial.begin(115200);
  delay(10);
  wifi_sniffer_init();
  pinMode(LED_GPIO_PIN, OUTPUT);
  WiFi.mode(WIFI_MODE_STA);
  Serial.println(WiFi.macAddress());


  WiFi.mode(WIFI_STA);
  // dataQueue = xQueueCreate(10, sizeof(struct_message));
  // Init ESP-NOW
  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }

  // Once ESPNow is successfully Init, we will register for Send CB to
  // get the status of Trasnmitted packet
  esp_now_register_send_cb(OnDataSent);


  // Add peer

    memcpy(peerInfo.peer_addr, broadcastAddress2, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;

  // Add peer
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
  }

}


void loop() {


  if (digitalRead(LED_GPIO_PIN) == LOW)
    digitalWrite(LED_GPIO_PIN, HIGH);
  else
    digitalWrite(LED_GPIO_PIN, LOW);
  vTaskDelay(WIFI_CHANNEL_SWITCH_INTERVAL / portTICK_PERIOD_MS);
  wifi_sniffer_set_channel(channel);
  channel = (channel % WIFI_CHANNEL_MAX) + 1;

  //esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &drone,
sizeof(drone));
}
```

End of Sensor Code

Satellite/Receiver Code:

```
#include <esp_now.h>
#include <WiFi.h>

#include <string.h>
#include <esp_system.h>
```

```
#define ALERT 5
// Define the MAC address for broadcast
uint8_t broadcastAddress[] = {0xDC, 0xDA, 0x0C, 0xFE, 0xA4, 0xE0};


typedef struct {
    char a[32];
    int b;
    int c;
    int d;
} struct_message;

struct_message incomingReadings;
struct_message myData;

void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len)
{
  Serial.println("Data received.");
  memcpy(&myData, incomingData, sizeof(myData));
  Serial.print("Bytes received: ");
  Serial.println(len);
  Serial.print("Message: ");
  Serial.println(myData.a);
  Serial.print(myData.b, HEX);
  Serial.print(":");
  Serial.print(myData.c, HEX);
  Serial.print(":");
  Serial.println(myData.d, HEX);
  if (myData.d > 0){
    analogWrite(ALERT, 51*myData.d);
  }
  else{
    analogWrite(ALERT, myData.d);
  }
}

void setup() {
  Serial.begin(115200);
  delay(1000);
  WiFi.mode(WIFI_MODE_STA);
  Serial.println(WiFi.macAddress());



  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }
  esp_now_register_recv_cb(OnDataRecv);



}

void loop() {




}
```

8.3 *Component Data Sheets*

For components purchased from Amazon no datasheet was available, but a link to the component has been included.

- ESP32 C3 WROOM-02
- CP2102 USB to UART
- First microphone we purchased (lower quality)
- Second microphone we purchased (slightly higher quality)
- Vibration Motors
- RGB LED
- Voltage Regulator

All other components (diodes, capacitors, resistors, MOSFETs, and connectors) are stock components from the Innovation Lab.