Magic Door Sensors

DESIGN DOCUMENT

Team 09
Daji Qiao (EE, ISU)
Team Members/Roles

Ryan Lanciloti - Report Manager
Ben Pierre - Webmaster
Abdelaziz Hassan - Power Systems
Chinar Kaul - Meeting Facilitator
Alyssa Marshall - Meeting Scribe and Team Manager
Jared Hermon - Test Engineer

sddec21-09@iastate.edu
https://sddec21-09.sd.ece.iastate.edu

Revised April 17th, 2021 Version 3

Executive Summary

Development Standards & Practices Used

- Circuit
  - US power circuit wiring color codes
- Hardware
  - CISPR and IEEE Electromagnetic Interference Standards
- Software
  - PEP 8 and Arduino Styling Guides

Summary of Requirements

- Detects a door's rotation at 15-degree angles.
- Detects a door state from 10ft away (with the line of sight)
- No active sensors on the door
- Arm and disarm the system
- Notify user within 5 seconds of a door opening

Applicable Courses from Iowa State University Curriculum

- Com S 573
- Com S 228
- Com S 309
- CPR E 288
- CPR E 281
- CPR E 488
- CPR E 489
- E E 417
- E E 230
- E E 201
- E E 311
- E E 332
- ENGL 250
- ENGL 314
- STAT 330

New Skills/Knowledge acquired that was not taught in courses

- Wifi CSI Packet collection
- ESP32 programming

Table of Contents

1 Introduction **5**

Acknowledgment 5

Problem and Project Statement 5

Operational Environment 6

Requirements 6

Intended Users and Uses 6

Assumptions and Limitations 7

Expected End Product and Deliverables 7

Project Plan **7**

2.1 Task Decomposition 7

2.2 Risks And Risk Management/Mitigation 8

2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria 8

2.4 Project Tracking Procedures 8

2.5 Personnel Effort Requirements 9

2.6 Other Resource Requirements 9

2.7 Financial Requirements 9

3 Design **9**

3.1 Previous Work And Literature 9

Design Thinking 10

Proposed Design 10

3.4 Technology Considerations 11

3.5 Design Analysis 11

Development Process 12

Design Plan 13

4 Testing **16**

Unit Testing 16

Interface Testing 16

Acceptance Testing 16

Results 17

5 492 Implementation 17

5.1 Antenna Implementation 17

5.2 Hardware Implementation 18

5.2.1 Access Point ESP32 18

5.2.2 Station ESP32 18

5.2.3 Relay ESP32 18

5.3 Server Implementation 19

5.4 Machine Learning Implementation 19

5.5 Front end Implementation 20

6 Closing Material 21

6.1 Conclusion 21

6.2 References 22

6.3 Appendices 22

6.3.1 Operations Manual

List of figures/tables/symbols/definitions (This should be the similar to the project plan)

Figures:

- Section 2.4 - Figure 1: Gantt Chart
- Section 3.3 - Figure 2: Design Sketch
- Section 3.7 - Figure 3: Design Plan
- Section 3.7 - Figure 4: App Design Mockup (Screens 1-3)
- Section 3.7 - Figure 5: App Design Mockup (Screens 4-6)
- Section 3.7 - Figure 6: Model Training Mockup (Screens 1-3)
- Section 3.7 - Figure 7: Model Training Mockup (Screens 4-6)

Tables:

- Section 2.6 - Table 1: Time estimation for each task

Definitions:

- **ESP32** - highly-integrated and low-powered microcontroller with inbuilt antennas and added Wifi Module
- **CSI** - "This information describes how a signal propagates from the transmitter to the receiver and represents the combined effect of, for example, scattering, fading, and power decay with distance" *(Channel state information 2020)*
- **Antenna Design** - a system of multiple antennas and a reflector that is used to transform and reflect radio frequency (RF) signals
- **ML** - Machines learning - A subset of artificial intelligence

1 Introduction

1.1 Acknowledgment

We would like to acknowledge and thank our facility advisor and client, Dr. Daji Qiao, for all of his help throughout this project. He provided us with information on the different technologies we could utilize and provided feedback on a weekly basis for where to go next. We would also like to thank Dr. Andrew Bolstad for his CSI and antenna design input. We would like to thank Dr. Mani Mina and Ph.D. Student Wei Shen Theh for their continuous input and guidance for electromagnetic questions.

Additionally, we would like to thank Steven M. Hernandez and Jonathan Muller for access to their open-source GitHub repositories. Access to these templates has made development significantly more efficient [5],[6].

1.2 Problem and Project Statement

Picture: You're driving home from the airport after a long trip. Before leaving, you installed this new security system that utilizes a wireless door sensor powered off of a 9-volt battery and interfaces with a base station somewhere else in the room. As you're pulling up to your house, you notice the door is slightly ajar. You approach the door curious; you're confident you closed it before you left. The security system would have alerted you if someone had broken in, right? Well, the TV, latest-generation gaming console, and most of your valuables are gone. Panicked, you check the door sensor to see what's wrong and notice the light is off. After some tests, you realize the battery you put in your sensor was already partially drained, and it died only moments after you walked out of the door for the trip. If only there were a modern, non-intrusive, wireless, batteryless security system that was easily affordable, attractive, and easy.

Introducing the Magic Sensor, we are proposing a wireless sensor that doesn't utilize a battery of any sort. Modern security systems with a physical sensor attached to doors require a battery that needs to be changed every so often. Checking and replacing the battery adds hassle to the system and can prove extremely problematic if you're away and you get alerted that a sensor has died.

Our solution is to use Channel State Information (CSI) included within WiFi to track the movement of a door to determine if it's opened or closed. Using one transmitter and one receiver to send and receive information about the angle and time of flight of the WiFi signals as they propagate through the room. A reflective piece of aluminum on the door deflects the signals differently depending on if the door is open or closed.

Our project aims to improve upon solutions to the age-old problem of security in the modern age. WiFi is cheap, and the technology is very well established, so we aim to solve an old problem with modern solutions.

1.3 Operational Environment

Our product is expected to be used inside of a building, away from most harsh environments. Our sensor must work on any arbitrary door; that could be an interior door and an exterior door of the building. The system consisting of the base station and door sensor is expected to not face any outdoor elements. With that statement, it does not need to be waterproof, withstand large impacts, or handle extreme temperature changes. As such, it must work under a semi-wide range of temperatures, and we chose between -10° C and 40° C inclusively.

1.4 Requirements

Functional:

- Detect if a door is open o r closed with 85% accuracy with false-positive reporting around 5% to 10% and false-negative reporting less than 5%
- Detect a door's state up to 10 feet away from the base unit
- No powered no wired sensor on the door
- Alert client via an application on their phone within 5 seconds if a door opens or closes
- Capable of arming and disarming system

Non-Functional:

- $300 budget for total system
- The door module must be less than $30 to allow a user to add more sensors to their system in the future
- UI must conform to current design stylings
- The door module must weigh less than one pound and be less than 4" by 4" when installed

1.5 Intended Users and Uses

This product will be deployed by a homeowner/business owner/property owner who wishes to have a non-powered, tamper-resistant system of detecting door states. This user wants to have the system installed by a professional and then never touch it again. The only interaction needed between the user and the system is arming and disarming the system via the application on their phone.

1.6 Assumptions and Limitations

Assumptions:

- The system will only be used inside
- Doors will be within 10 feet of the base station
- The building will have WiFi so that the ESP32 is capable of relaying data to the server
- Expecting the user to understand English as our application will be only written in English
- The user owns an Android phone (or other internet connected device)

Limitations:

- The door module will be no more than 4” x 4”
- The door module will not exceed more than 1 lb

1.7 Expected End Product and Deliverables

- Security Hub
  - A suite of connected ESP32s
- Web App
  - Reachable via the internet, capable of reporting information about the door
- Server
  - Capable of being deployed on any Linux box, it only needs port forwarding to set up.

2 Project Plan

2.1 Task Decomposition

- Develop Door Module
  - Analyze room layout
  - Develop door system layout
  - Test aluminum door module with ESP32s
- Develop ESP CSI Harvesting Network
  - Develop Code for ESP32s
    - Refactor original code
    - Establish harvesting routine
    - Allow multiple channels
  - Interface with a remote server
- Develop Inferencing on Server
  - Feed data from ESP32s
    - Interface with ESP32s
    - Establish a protocol for handling several streams
  - Generate prediction as to door state

            - Feed normalized CSI ESP data into TensorFlow or equivalent
    - Interface with GUI
- Develop user GUI
    - Interface with sever
    - Allow arming and disarming
    - Setup notifications upon state change
    - Polish UI if time allows

2.2 Risks And Risk Management/Mitigation

- ESP CSI Harvesting Network
    a. Packet unreliability was mitigated with multiple sensors and server side checks
    b. Sensors could be tampered with and as such will be put in tamper-proof boxes
- Develop Inferencing on Server
    a. Machine Learning will vary by setup, as such we included software to train the system
    b. The server could be compromised, as such, we implemented passwords and used Iowa State's built-in protections
- Server Reliability
    a. Should the server go down for an unforeseen reason CRON jobs will reboot it, or alert our backend team

2.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

1. Develop a concrete design for our system as a whole
2. Put together functional code on our ESP32s
3. Parse through data from ESP32s into human-readable output
4. Pass data form front end to backend
5. Develop a machine learning algorithm that can interpret and train on data from ESP32s
6. Achieve an inference accuracy of 70%
7. Improve inference accuracy to 85%
8. Design and develop a way to retrain the model efficiently

2.4 Project Tracking Procedures

Our group utilized Github for keeping track of all software we used for this project. We made use of Google Drive for the sharing and editing of all documents which we have created for this project. Once we were able to develop a solid plan for how to execute this project, we used Trello to keep track of features and milestones in development.

2.5 Personnel Effort Requirements

Table 1: Time estimation for each task.

Task	**Number of hours**
Writing ESP 32 Code	15
491 presentations and documentation	16
Researching	24
Establish machine learning on ISU servers	18
Develop GUI	15
Develop Aesthetic Front End	10
Link systems via MQTT	10

2.6 Other Resource Requirements

- 3 ESP32
- Aluminum sheet for door module

2.7 Financial Requirements

We do not have a concrete budget for this project, however, our client has specified that our final iteration must cost $300 or less for one base station and one sensor, connected to our server. The cost per additional sensor must be less than $30.

3 Design

3.1 Previous Work And Literature

From the research we have done, there currently does not appear to be a wireless home security system that utilizes CSI for detecting door states. The current door sensors on the market either

require the physical sensor to be wired in or require a battery to operate and use a reed switch for detection. This project is both riveting and daunting. It is riveting because we are essentially researching the technology and its applications as it is still an abstract technology. However, it is daunting because, going into the project, we have no idea about the feasibility of CSI in a real-life security application. We are doing the R&D and experiments to figure out if we can develop a system that can detect the state of a door. For the relevant documents, please see section 6.2.

3.2 Design Thinking

There were various ideas that were brought up during the ideation process. One potential idea was imaging processing using OpenCV and Tensorflow. However, we did not proceed with this idea as it did not fit our client's needs and there was privacy concern. We also considered adding a servo to the hinge of the door to generate a current that would power a relay when it was opened, but this was deemed too invasive. Additionally, we allow the user to install this product themselves, which would not be possible with this approach. Our final plausible idea was to put a pressure-sensitive switch on the door that would relay data to a base station, however this idea was also deemed too invasive. We thought about using an RF harvesting system with a hall effect sensor on the door, but this was deemed too complex to complete in our necessary timeline.

Once we determined CSI was the best system to use, we generated multiple ideas for the door module, outlined below.

3.3 Proposed Design

There are two possible methods of design: one involves CSI packets with a constant feed of phase and amplitude values, and the other method is putting a pressure-sensitive sensor on the door that emits a pulse when the door changes.

Upon suggestion from the client, we have opted to take the CSI approach, and have started testing this approach with a blend of open source code and our homegrown code. The system is composed of the ESP32 acting as a transmitter, an additional ESP32 functioning as a receiver, and a reflective piece of aluminum on the door in a triangle formation. The input of the system is the state of the door, and the output of the system is an analysis based on the door state resulting in an "open" or "closed" decision. The movement of the reflective piece of metal is used to create a change in amplitude to the RF waves that our CSI system will be able to detect and measure. The CSI data will be analyzed by a deep learning algorithm to accurately detect the door state. This whole system will be able to fulfill our functional goals of accurately detecting the door state with low false-positive and false-negative readings.

The specific metal chosen for the door module was aluminum as it fulfilled the functional requirement of not having an active door sensor but still helping provide the door state due to its high reflection ability. Additionally, aluminum filled all of our non-functional requirements for this project. Aluminum was one of the most cost-effective choices when considered against copper. Additionally, aluminum was chosen as it does not rust over time compared to steel, which was

another considered choice. Aluminum provides the minimum alteration to the appearance of the door and fits our weight and size requirements.

As detailed in our functional requirements and the diagram below, the base station can be up to 10 feet away from the door.

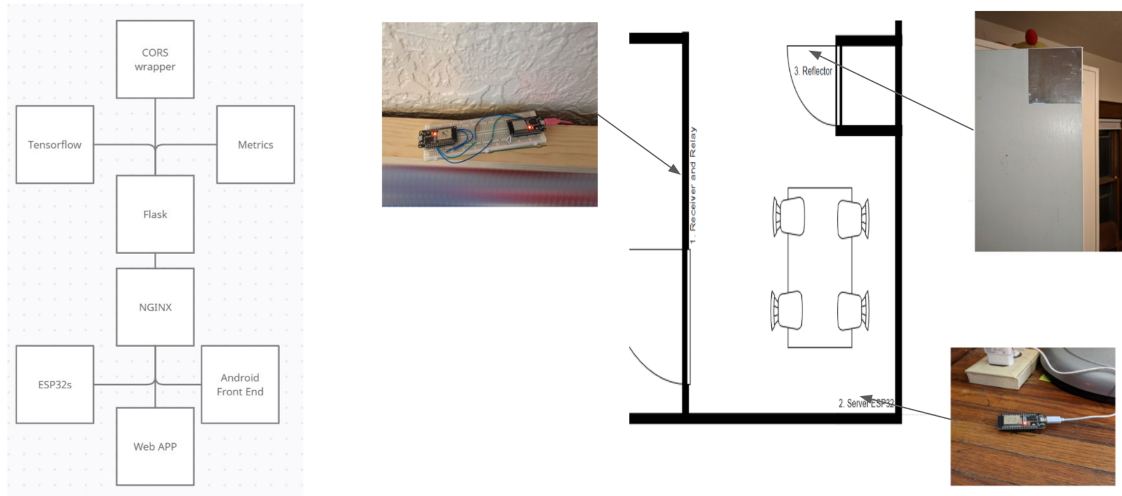


Figure 2: Implemented Design Sketch

3.4 Technology Considerations

One big issue we are running into is that thus far only one consumer-level microcontroller is available to analyze CSI packets, and to the best of our knowledge only one or two papers exist that actually implement this technology. Additionally, CSI Preambles seem to be notoriously unreliable and thus require several sensors for redundancy. The upside of this technology is that they are cheap and very easy to duplicate should the need arise.

Other solutions, such as placing an RF harvesting circuit on the door would increase reliability and deployability. However, they would negatively affect the client's vision and would require an unaesthetic implementation of the sensor design on the door.

3.5 Design Analysis

Our proposed design proved somewhat effective upon implementation. Our core systems, apis, and components all operate correctly. We saw successful propagation start to finish of the expected signals. We did not see a satisfactory level of accuracy or speed, however we outlined mitigations to these risks in section 2.2, as well as 5.5.

3.6 Development Process

As a group, we are familiar with Waterfall, Agile, and Test-Driven Development (TDD). From what we understand, Waterfall is a linear form of development and the least iterative out of the three. With Waterfall, we would gather requirements, design a solution, test, and deliver a final product. A major problem with Waterfall development is that if during the later stages of development, a bug is found or if a client changes the project requirements, engineers would have no way of integrating changes in the project unless they completely start over.

On the other hand, Agile is an iterative development process that encourages team collaboration and quick deliverables. The benefit of Agile is that it accounts for flexible client requirements. Agile teams are structured and include roles such as Scrum Master and Product Owner. Additionally, Agile teams work in a structured timeline. For example, teams have a Daily Standup every day during the sprint, then reflect on the past and future activities in the Sprint Review, Sprint Retrospective, and Sprint Planning meetings.

Test-Driven Development tightly integrates testing into the development process. A new test is written every time new technologies are introduced to guarantee every part of the project is thoroughly tested. A key feature of TDD is developing as little of the project and as little of a unit test as possible during each iteration to ensure that no part of the project goes untested.

Our team opted to use a flexible version of agile, with weekly Scrums, and biweekly client meetings. This served us well and allowed for rapid progression, as well as provided a natural sync point for when a team member was having issues and required support.

3.7 Design Plan

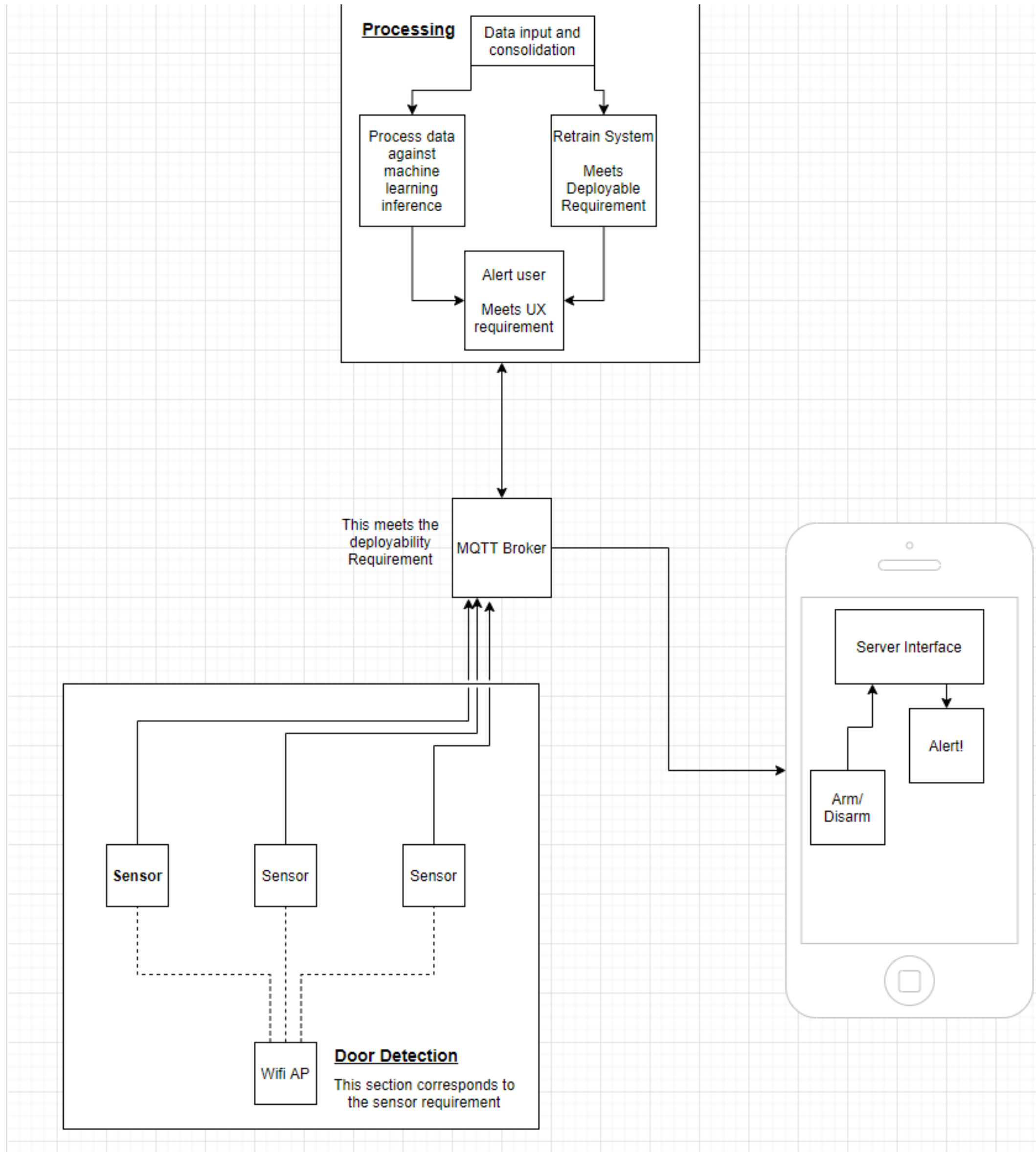


Figure 3: Design Plan

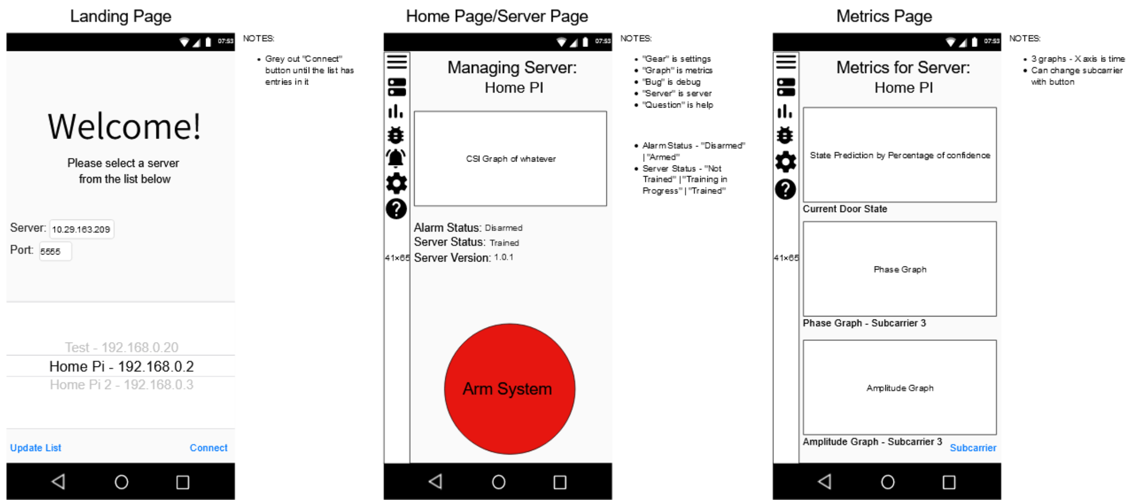


Figure 4: App Design Mockup (Screens 1-3)

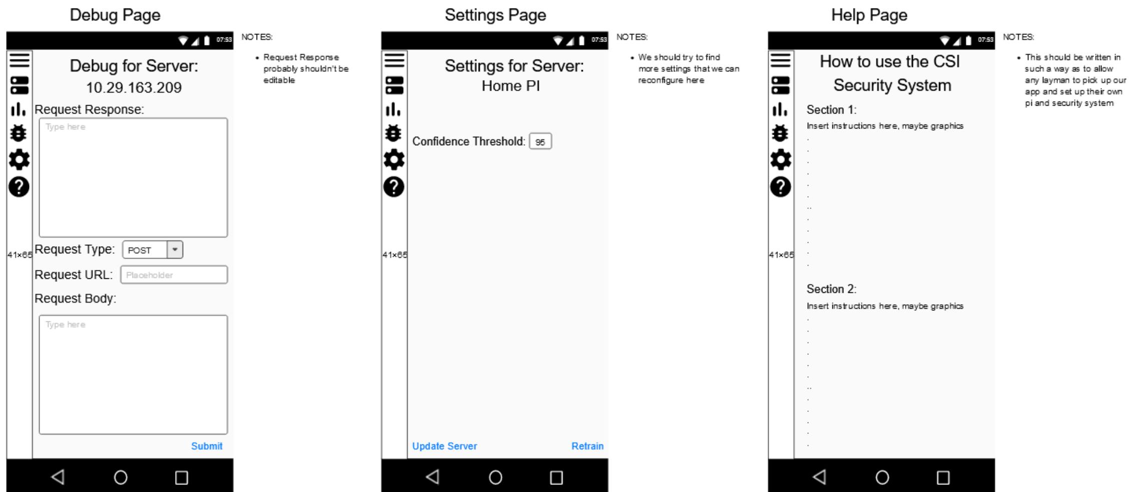


Figure 5: App Design Mockup (Screens 4-6)

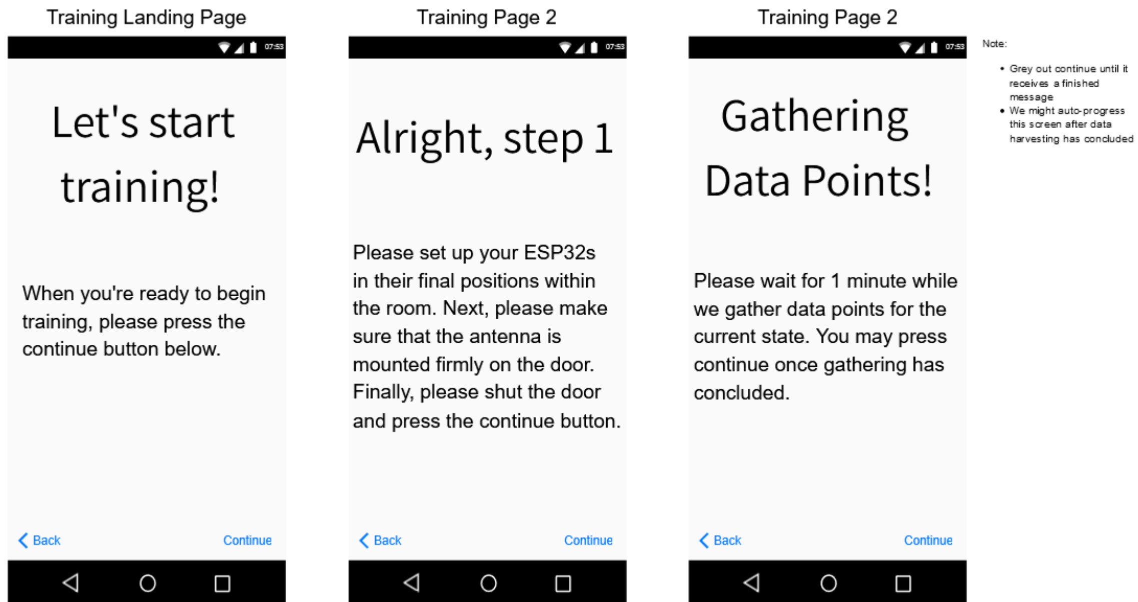


Figure 6: Model Training Mockup (Screens 1-3)

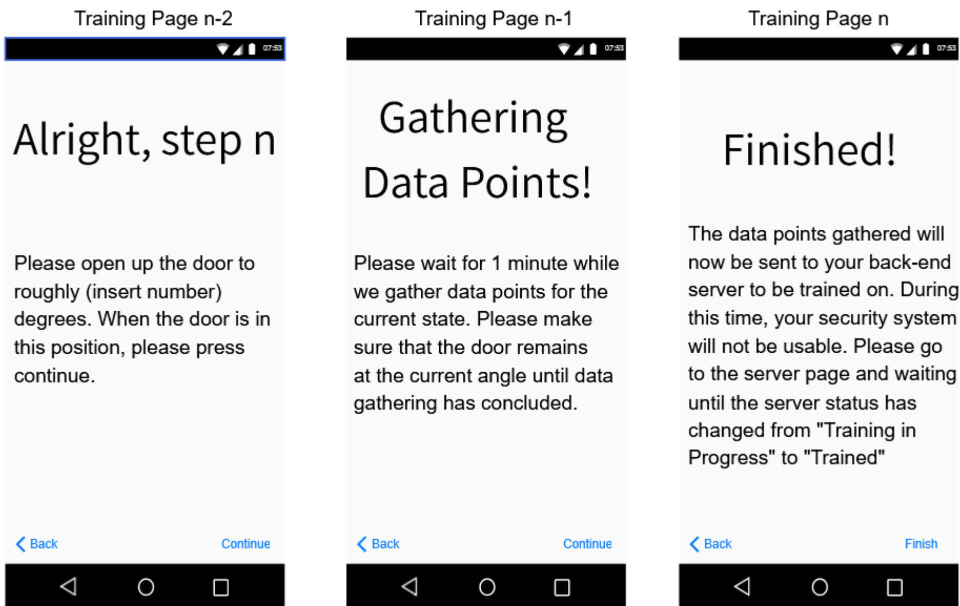


Figure 7: Model Training Mockup (Screens 4-6)

This design meets all of our functional requirements, while also breaking the system into smaller modules that can be built and tested individually.

4 Testing

4.1 Unit Testing

Currently, we are doing unit testing for our ESP32s, backend API, and GUI. One ESP32 establishes an Access Point (AP) and the other ESP32 connects to said Access Point; thus both subsystems must be tested. Unfortunately, the two are intertwined, and as such, both must be tested together.

Additionally, we test the component that handles our ESP32 link between the ESP32s and our server. Our server is located on Iowa state's network, while our tools are external. As a result, ISU ETG has set up a reverse proxy to allow us to forward our data onto the network securely. This link requires flow testing. To accomplish this we use a 'ping' like system where we send packets and measure the successful delivery percentile.

Another component that must be tested is the effectiveness of the door module. This component can be tested individually once we have meaningful CSI data implemented with the ESP32s. The various designs consisting of different materials were considered. With research, aluminum was deemed the best material as it reflects the RF waves most effectively for our purpose. Additionally, we have landed on a 4" by 4" flat piece of aluminum. We test a parabolic shape instead of a flat shape to see if we can maximize the CSI system's efficiency.

The final component that needs to be tested is our machine learning system. This component receives packets from our python relay and interprets them to determine door states. This component can be tested individually with constant inputs to verify we generate the expected information.

4.2 Interface Testing

We used one interface in our system. This interface took the form of a flask API on our ETG server. We use this interface to handle all multisystem communication. This includes ESP32 mesh to server, server to machine learning model, model to server, and server to front end.

We can test the ESP32 portion by validating we are seeing responses with code 200 on the ESP32s and validating that for each post to the end point our cached CSI data changes.
We test the server/ML bridge by validating that when we call the ML model with any data we see a success flag go high, and a valid output seen.
We test the front end / server link by validating that on a request we get a packet with our CORS key, in a format we expected (as dictated by our JSON parser).

4.3 Acceptance Testing

We prove acceptance testing by setting up a demonstration to our client in which we have the entire system running and set up. We demonstrate that opening and closing the door does update

the state on the GUI application. This proves functional requirements have been met. Our non-functional requirements are easy to demonstrate as they consist of visible features such as the sensor is less than 4 by 4 inches in size. These non-functional goals are obvious at a demonstration, as mentioned earlier.

4.4 Results

The non-functional requirements were met in this project. The door module, the aluminum reflector tape, was only seven dollars which was less than the thirty dollar budget. The door module weighed less than one pound and was less than 4" by 4" when installed. This door module met the requirements of being non-invasive and provided minimum disruption to the design of the door. The reflector team ran tests to verify that the reflector tape would not peel any material off from the door meeting design requirements of a consumer product.

The demonstration of the system was done by opening and closing the door and watching the update of the state on the GUI application. After the integration of the system, our machine learning model was reporting around fourteen percent accuracy.

Since our integration of our machine learning model had some difficulties, our results will be based on the machine learning model when run locally with our CSI data.

After the second round of machine learning model modifications, we ran the acceptance testing again. This time we saw results of about 45% accuracy, with a peak near 60%. Admittedly this is not near our requirement of 85%, but with more testing and future integration of principal component analysis, we expect to hit that benchmark.

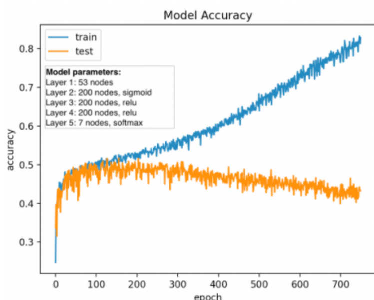


Figure 9: Model accuracy with diverging training and testing accuracy, indicating overfitting

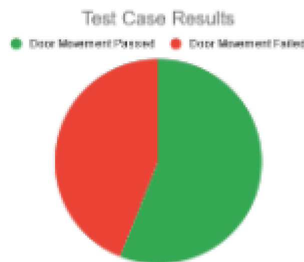


Figure 8:Test Suite One Results

5 492 Implementation

5.1 Antenna Implementation

The final design the team decided to move forward with was not an antenna, but a reflector. This reflector is a 3 inch by 3 inch square of aluminum mounted to the top of the door, on the side with the rest of our system, on the corner farthest from the hinge. We opted for this material as it is

highly reflective towards RF waves, is cheap, lightweight and easy to shape. We opted for the given location as it provides the most dynamic range of interference as the door moves.

5.2 Hardware Implementation

Our hardware implementation involves three different ESP32s.

5.2.1 Access Point ESP32

Our first esp32 sits in a corner and broadcasts an access point to the world. This access point functions as a UDP relay. When it receives a packet it simply replies back to the sender with a duplicate message. We opted to use an ESP32 for this as the esp32's were the lowest cost microcontrollers we could find that were easily programmable and supported CSI through their built-in Wireless chips. The code for this was written in C with the ESP-IDF extensions, and heavily inspired by [5] "ESP-IDF" Jonathan Mueller.

5.2.2 Station ESP32

This ESP32 was responsible for most of the work involved in the hardware. It generates a packet as fast as possible and sends it towards the access point ESP32. It then waits for a response. Upon receiving this response it extracts the CSI preamble, parses out the valid subcarriers, and calculates the phase and magnitude of each subcarrier. When this is complete it writes it to a UART link, and then restarts the process. Similar to the Access Point ESP32 we opted for this controller as it was a low cost, accessible microcontroller that has access to CSI packets. We again opted to use C code with the ESP-IDF as it exposed the CSI api for us.

Note: This code has several levels of verbosity defined for debugging, visual expression of packet receiving, optimizing for UART vs Console writing. This is also controlled via define statements.

Note: During the design phase we had much back and forth about where to do the phase/magnitude calculations, if we wanted to include RSSI data in the UARTstream, etc. After the second change in opinion the hardware engineers got tired of reworking the code, so the functionality to calculate phase, magnitude, RSSI, and raw values are defined by compiler level defines, allowing for rapid deployment and easy changes as need arises.

Note: The access point has all the same capabilities of this ESP32 (data parsing and UART) however they are disabled via defines.

5.2.3 Relay ESP32

This ESP32 is connected to the Station ESP32 via a UART bridge with a baud rate of 115200. This rate offers high data transfer while maintaining accuracy of the data transfer. When this receives a full datapoint, as designated by a newline, it packs it into a post request and sends it to our backend

server. It is able to do so via it's wireless internet connection to our home network. This ESP32 blinks a blue onboard LED each time it makes a post, regardless of the success of that post, to indicate that it is receiving and sending. This allows for easy debugging when something stops working. We opted to use an ESP32 for this sensor for a unified infrastructure, however it is programmed C via arduino as the arduino libraries provide much of the functionality required, and we did not need access to the CSI data on this device.

5.3 Server Implementation

The brain of our solution is a backend server running on Iowa State's intranet. This server is running two services, an apache web server that's hosting the documentation for the python modules. This documentation is auto-generated when the server is redeployed. The second service is a flask application which acts as an API for both the ESP32s and the end user to interact with the system. The user can query information from the API such as model status, current door state, and request the model be retrained. The ESP32s send data points to the backend server which automatically have inference done on them so long as the model is in a trained state. This allows the end user to request the current door state and for the data to already be processed. Also, it would allow for long term logging to be done of the door state in the event that data analytics should be done later on.

Additional functionality added to the backend API is the ability to call an endpoint in order to automatically pull down the most recent version of the code and relaunch the server. There is a daemon that checks to see if the current process is running and if not, it'll pull down the most recent version of the code and attempt to relaunch the application. This means that once the server is deployed, there should be little to no downtime. If a breaking change is pushed and the server crashes, once the fix has been pushed to github, the daemon should update the code on the server and relaunch with the fix.

Also, while not the server itself but an aspect of the system as a whole, is the docker container created to support running the server. We rely heavily on a lot of packages and python modules not natively integrated in the standard python distribution and the docker container allows for any person to simply download and run the entire backend with little to no worry about how it will run on their system.

5.4 Machine Learning Implementation

The machine learning team settled on using a neural network model after considering several other models. Initially, we considered using a linear regression model with the assumption that the incoming CSI data would be reduced down to one dimension, allowing the model to predict based on the relationship between CSI data and door angle. However, we realized that our dimensionality

reduction plan, PCA, would reduce the data to a minimum of 53 dimensions, which would not be feasible to input into a linear regression model.

With this in mind, we switched to using a neural network model to handle the multiple dimensions of the input data. A neural network would be able to learn the complex relationships between the multi-dimensional data and the door angle using several hidden layers and nodes during the training session. The neural network was designed to input 53-dimension CSI data and output a prediction falling under one of 7 categories corresponding to the door angles at 15 degree increments. The model also can output a confidence interval, detailing the confidence that the current input data corresponded to each of the 7 angle categories.

Several different model parameters were tested. The final model state was as follows: 1 input layer with 53 neurons, 2 hidden layers with 200 neurons each, 1 output layer with 7 neurons running 400 batches and 250 epochs. Respective to the model layers, the following activation functions were used: sigmoid, relu, relu, softmax. A categorical cross entropy loss function was used to handle the multiple output labels. An Adam optimizer was used since it is generally accepted in the machine learning community that Adam provides the best performance as an adaptive optimizer.

During implementation, there were discussions on the type of data fed into the model. In an effort to feed only real numbers into the machine learning model, we made the decision to convert the CSI data from imaginary numbers to phase and magnitude representations. This decision brought up the question of how to feed phase and magnitude into the model. Though phase and magnitude are both real number data, they use two different units, so the data cannot be fed into the model at the same time assuming the units are equivalent. Since both phase and magnitude combined represent CSI data, we decided to make two instances of the same model, one model for phase and one model for magnitude. We hypothesized that the predictions from these two models would be the same, since the training data for both models came from the same input data and both models were predicting from the same set of new data.

5.5 Front end Implementation

Our front end switched from an Android application to an HTML webapp mid semester. This decision was made to cut complexity from our project, and because the ESP32 engineer was done with ESP32 engineering and available to work on the front end, and was much more familiar with HTML, allowing for a faster development cycle.

The front end consists of a few screens, listed and shown:

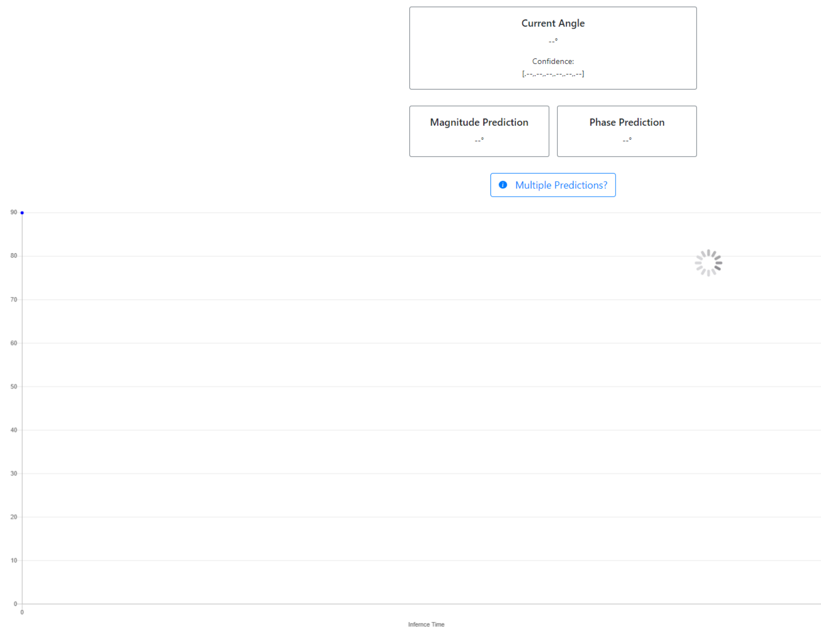
- Arm/Disarm : Allows the end user to turn on or off the system

  The server is currently : Armed

  Disarm

-

- Prediction : Outputs the predicted values of both models, calculates the agreed upon angle, and graphs prior angles.



- Insights : Forwards the CSI data to the front end for viewing / debugging, providing useful insight as to the inner state, and a graphical interpretation of the CSI data.

These screens are available on our website, under the project tab.

Note that our android application still exists in an incomplete form, and can be accessed via our github.

6 Closing Material

6.1 Conclusion

At the start of the semester we had taken the previous groups project, recreated it, and determined it to be unpromising. We had then designed and began to implement our own plan for a more complete, more correct system. This semester we completed the implementation of this project, and moved onto testing. We were able to demonstrate a complete system, with data parsing from start to finish, however we had unsatisfactory accuracy and speed. We were unable to correct these issues, but we were able to lay out plans to improve upon these issues, which can be provided to future students.

6.2 References

[1] Al-Qaness, M. A., Li, F., Ma, X., & Liu, G. (2016). Device-free home intruder detection and alarm system using wi-fi channel state information. International Journal of Future Computer and Communication, 5(4), 180-186. doi:10.18178/ijfcc.2016.5.4.468

[2] Brescia, F., Gringoli, F., Brescia, U., Darmstadt, M., Schulz, M., Darmstadt, T., . . . Authors: Francesco Gringoli University of Brescia. (2019, October 01). Free your CSI: A Channel state information EXTRACTION platform for MODERN Wi-fi Chipsets. Retrieved March 04, 2021, from https://dl.acm.org/doi/abs/10.1145/3349623.3355477

[3] Ma, Y., Zhou, G., & Wang, S. (2019). WiFi sensing with Channel state information. ACM Computing Surveys, 52(3), 1-36. doi:10.1145/3310194

[4] "Channel state information," Wikipedia, 18-Aug-2020. [Online]. Available: https://en.wikipedia.org/wiki/Channel_state_information#:~:text=In%20wireless%20communications%2C%20channel%20state,and%20power%20decay%20with%20distance. [Accessed: 03-Apr-2021].

[5] "ESP-IDF" Jonathan Mueller[Online]. Available : https://github.com/jonathanmuller/esp-idf

[6] "ESP32 CSI Toolkit" Steven M. Hernandez[Online]. Available:
https://stevenmhernandez.github.io/ESP32-CSI-Tool/

[7] H. Guan and D. D. L. Chung, "Absorption-dominant radio-wave attenuation loss of metals and graphite," Journal of Materials Science, vol. 56, no. 13, pp. 8037–8047, Jan. 2021, doi: 10.1007/s10853-021-05808-2.

6.3 Appendices

6.3.1 Operations Manual

Front end

1. Code upload
    a. To install code on ⅔ of the code you will need to install the espidf tool kit. This information can be found here
        i. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/
    b. The third esp32 will require you to have an Arduino IDE installed. The default IDE can be found here:
        i. https://www.arduino.cc/en/software
    c. You will also need to clone the github repository (Magick Sensors Reimagined) to a local directory.
    d. Access Point ESP32
        i. Plug in an ESP32 to your computer.
        ii. Open the esd-idf command line tool and navigate to the directory 'ESP32 Code/active-ap'
        iii. Run the following commands *

1. idf.py make
2. idf.py -p {YOUR COM PORT} flash
3. idf.py -p {YOUR COM PORT} monitor

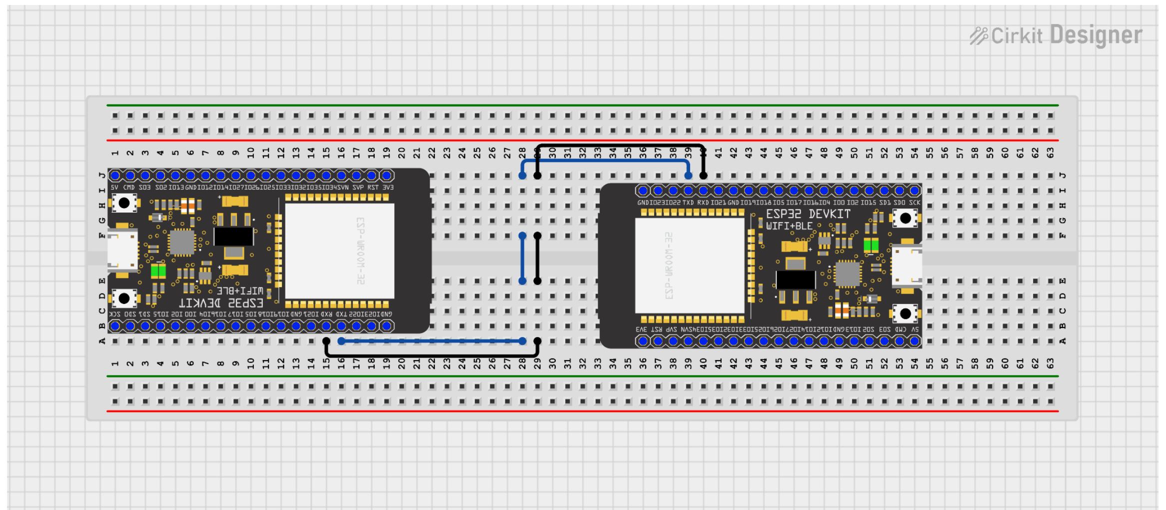
iv. If all is well you should see the ESP32 startup, output information about it's AP parameters, then state it is awaiting data.

e. STA ESP32
   i. Plug in an ESP32 to your computer.
   ii. Open the esd-idf command line tool and navigate to the directory 'ESP32 Code/active-sta'
   iii. Run the following commands *
      1. idf.py make
      2. idf.py -p {YOUR COM PORT} flash
      3. idf.py -p {YOUR COM PORT} monitor
   iv. If all is well you should see the ESP32 startup, and then indicate it is successfully connected to the AP.

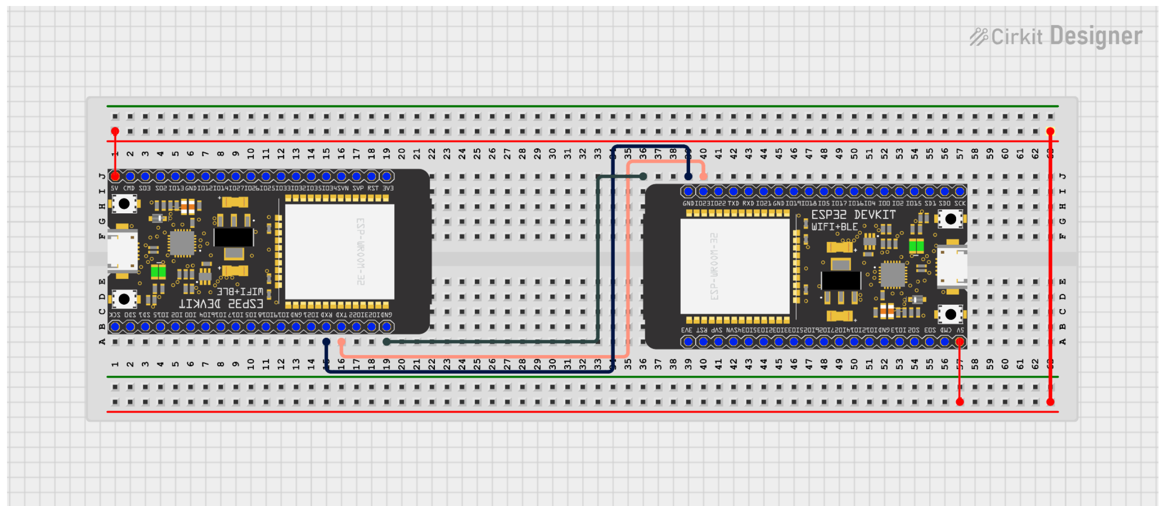
f. Note : * Indicates that when running the flash command, if you have an older ESP32 you may need to hold the boot button while running flash. This is indicated by the console hanging on 'Connecting to board'

g. UART Bridge ESP32
   i. Plug in an ESP32 to your computer.
   ii. Open the Arduino IDE of your choosing and open Arduino IDE Code / SerialPassthrough / SerialPassthrough.ino
   iii. Change the values of 'SSID' and 'PASSWORD' to be the information of a 2.4ghz wifi network with access to the internet.
   iv. Flash this to your esp32. Instructions on how to do this can be found on your IDE's resources.
   v. Open a serial monitor and restart your ESP32 by pressing the en button. You should now see information about the ESP32 connecting to the internet, a check to be sure it can get the time, and then a waiting status.

2. ESP32 integration
   a. The UART Bridge ESP32 and the STA ESP32 must be wired together to allow for the UART communication. Connect the TX and RX pins on your ESP32 to the RX and TX pins on the other ESP32, a diagram of this follows:



b. Optionally you can wire the 5v and gnd pins together to allow the use of one power source to power both the Relay ESP32 and the station ESP32. To do this follow the following diagram:



3. With the above steps completed and power applied to all three ESP32s the blue LED on the UART ESP32 should be blinking, and the red lights on all three ESP32s should be flickering. If not then there was a fault detected and you should move to section 6. Debugging.
4. Reflector
   a. The reflector should be placed on the side of the door facing the ESP32s, on the top corner of the door furthest away from the hinges.
5. Placement
   a. The placement of the ESP32s is not crucial, and can be adjusted to fit your aesthetic needs.
   b. Once the system is trained(see the information in the back end setup steps, the ESP32s cannot be moved.

c. For best results place the ESP32's so that the reflector is directly between the Access Point and the Station ESP32s. The closer to this the better.
d. Additionally the closer to the door the better, however we have had success up to 10 feet from each sensor to the door.

6. Debugging
    a. If you have issues with the hardware setup worry not.
    b. Start at your Uart Relay ESP32.
        i. Connect a serial monitor, ideally via the arduino IDE with a USB cable.
        ii. Set the isDebug flag high. Reflash the ESP32.
        iii. The ESP32 should now dump everything it reads over UART to the console. If you see nothing, move on to the next step. If you see valid numbers or random characters, ensure your UART baud rate is set to 112500 buad. If the problem persists redo your wiring between the two esp32s, it is likely you have a bad wire.
    c. Debugging the Station ESP32.
        i. With the ESP32 plugged in and the other two ESP32s powered open the ESP-IDF command line, navigate to the 'ESP32 Code/active-sta' folder within our repository and run the following command:
            1. idf.py -p {YOUR PORT HERE} monitor
        ii. Take note of the messages printing during start up. If you see errors about being able to connect to the AP please verify the SSID and PASSWORD fields are the same within the config of the AP and STA files. To check these navigate to the respective directories and run the following command
            1. idf.py menuconfig
        iii. If these fields are different please change them to line up, reflash this ESP32 and the server ESP32, and retry. If these are the same then navigate to the 'ESP32 Code / _components folder and open csi_component.h with a text editor of your choice, we use Visual Studio Code.
        iv. Change the verbosity flag on line 10 to be 3, save, and reflash the ESP32.
        v. Now running the monitor command should dump hundreds of numbers per second to the console. If these characters have defined chunks then set the verbosity flag back to 0 or 1, and reflash the ESP32. If this does not solve the problem then we know it is within the UART bridge.
    d. Debugging the Server ESP32.
        i. We have never had issues with this component and as such have no guidance.

Back End

1. Grabbing/Installing the code
    a. Start by cloning/downloading this Github repo: https://github.com/bjpierre/Magic-Sensors-Reimagined.git

        i. This Github repository should include all of the code required to run the backend
    b. Inside of the "Backend Server" folder is all of the code which supports the API service which runs on the server. Also included in said folder is a README file which contains information about each of the files, as well as instructions on how to run the backend. Those instructions will, more or less, be reiterated here
2. Method One: Install/Setup docker
    a. Provided in the Github repo which was listed above is a docker file that can deploy a container which contains all of the dependencies for the backend API. While not necessarily required, using the docker environment significantly simplifies deploying the backend API.
    b. After docker has been installed on the host machine, run the launch_docker script. This will probably require you, the end user, to first run `chmod +x launch_docker` followed by `./launch_docker`. This will pull down a prebuilt version of the container and run it.
    c. If any changes need to be made to the runtime environment of the docker environment, we have also included the Dockerfile used to create the container.
3. Method Two: Setting up the environment manually
    a. Assuming you, the user, want to avoid using docker or the docker vm is deemed to be too slow for the application, setting up the runtime environment should be relatively easy. First setup of an ubuntu runtime environment.
        i. This can be a virtual machine running on a bare metal hypervisor, or it could be a standalone machine. The only requirement is that it's routable to any machine that needs to communicate with it.
    b. Step 2: Install python 3.8
        i. Python 3.8 may be a soft requirement, however eventually you'll need to install tensorflow 2.6 which has a minimum python requirement. Python 3.8 is what we utilized and thus we know it's safe.
    c. Step 3: Install pip
        i. It may be possible to use another package manager, however we can't guarantee the packages we used will be available and up to date.
    d. Step 4: Run `pip install -r requirements.txt` from within the docker folder
    e. Step 5: Install/configure apache (optional)
        i. If you, the user, would like to host documentation on the backend API on the same server that hosts the API, then you'll need to install some sort of web server. We have ours configured to show all files within the `/var/www/html' folder so we didn't need to make a main traversal page. That configuration can be found in the `apache.conf` file within the docker folder. Also in this folder is the file which sets what port the apache server runs on, we have ours on port 20001.
    f. Step 6: Setup a cron job to run the server_daemon.py script every minute or so
        i. This is the script which will check to see if the server is running, and if not, it will download the most recent source code and relaunch the server. Not technically required but is a step towards automation.

4. Using and interacting with the backend
    a. Regardless of your method of setup, the server should now be accessible and ready to use. Installing postman and calling some of the debug functions within the API is probably the best way to guarantee that you're able to communicate with it. All of the code with in the "Backend Server" directory is documented, however I would recommend familiarizing yourself with Flask if you are struggling to understand how the `server.py` file works.