Ultrasense Smart Home

Using ultra-wideband technology to make the smart home smarter

---

Prepared by:

Peter Xiong

B.S. Computer Engineering

Jake Black

B.S. Computer & Cybersecurity Engineering

ECE441 Spring 2023 Lab 2 Group 2
Major Design Final Report
Instructor: Prof. Jafar Saniie
Illinois Institute of Technology
April 30, 2023

Table of Contents

Abstract

Over the past decade, the popularity of smart homes has increased exponentially, and has become a sixty-two billion dollar industry globally (Grand View Research). As the popularity of smart homes continues to grow, it is important to remember the basic principle of what a smart home should be. The purpose of a smart home should be to simplify daily tasks throughout the home.

As the number of smart home devices increases, and more technology throughout the home becomes IOT enabled, this principle of simplifying the home experience for the end user is becoming forgotten, and can sometimes complicate the user experience well beyond the regular inconveniences of a conventional home. Our system aims to simplify the smart home by introducing a positioning system into the home, allowing for smarter home automations, a simplified user experience, and a decluttered home app.

Introduction

This project started from a desire to improve the simplicity of the home experience. With the number of smart devices in a modern smart home these days, it is becoming difficult to keep track of all the devices, and even more difficult to control the desired device. In many cases, just using the light switch may be simpler for most people who don't want to reach for their smartphone each time they want to turn a light on.

This is where ultra wideband (UWB) technology comes in. Using UWB radios, we are able to determine the precise locations of UWB devices. This technology allows for much more precise location tracking compared to previous technologies such as wifi or bluetooth. By adding a home positioning system, the smart home now has more information than ever, and allows for smarter home automation, such as turning lights on and off when you enter and exit rooms. Automatically unlocking and locking doors as you leave and come home, allowing support for a smarter home app that shows the rooms closest to you, to reduce the amount of time you spend looking for the device you want to control in the home app, and allow for smarter home assistants that now can make smart inferences on the devices you want to control based on your location.

System Specifications

System Design

Our system uses four ultra-wideband radios in total. Three of the devices are used as the anchors, and are intended to be stationary. The fourth device is called the tag, and this device can move around freely. In our system, the tag polls all three anchors to determine the range from all three. It uses this data to then calculate its position relative to the anchors by triangulation.

Standards Used

WiFi (IEEE 802.11)

IEEE 802.11 uses radio waves to transmit data between devices. In our project, the ESP32 broadcasts this signal over its 2.4 GHz radio. It allows for wireless connection between the device and a router, and can communicate with any device on the local network, and connect to the internet. It is used to send WiFi UDP packets from the tag to the positioning server to give the server live positional data.

ESP Now

This protocol is created by Espressif, the company that designed the ESP32. It allows for communication between ESP32 devices. It also uses the same 2.4 GHz radio used for WiFi, but it allows communication between devices without the

need for a router. This protocol is used to communicate between the tag and the anchors, and allows the tag to send instructions for the anchors to execute, and allows the anchors to send data back to the tag.

System Constraints

Power

Our first constraint is power, as it's the foundation for everything. Since our system uses mobile devices, our limits & needs will follow the limits & needs of regular mobile devices. We want to reach 24+ hours of actual battery life to be able to operate through a full day covering even more than usual power activity. This way we can ensure a smoother user experience, that doesn't require recharging during the ideal times of usage.

Portability

Our next constraint is portability, because we're focused on tracking a person. For the situations where a user is unable to integrate our system with their device, we want to be able to provide a device that does not interfere with them. This means making the device & system as smoothly integrated as possible with not only their current smart home, or devices, but also how their physical person is while using our device. We'll aim to do this by minimizing the weight & any unnatural shape/size of the device.

Accuracy

Our next constraint is accuracy. As this is a positioning system, we want it to be accurate. Especially for our main application where we want to distinguish between the walls, floors, & distances of a house. As well as manage all the other uncertain factors that can be in play in the modern home. So we'll look to get within 6 inches of the actual coordinate, or position, of a device.

Ease of Use

Our final constraint is the ease of using our system. From the setup, to integration to daily use, in a physical & digital sense, we want our system to be easy to use. This helps everyone all day every day. Making the system easy to use, makes it easy & enjoyable for users to just use. It also makes it easy for them to actually understand what they have & what they're doing when they're doing it.

Hardware Design

Hardware System & Power

Our current system uses 4 ESP32 devices that get programmed differently, and a computer that can act as the positioning server. Below, in Figure 1, is a schematic of the devices & how they're generally connected in any application:

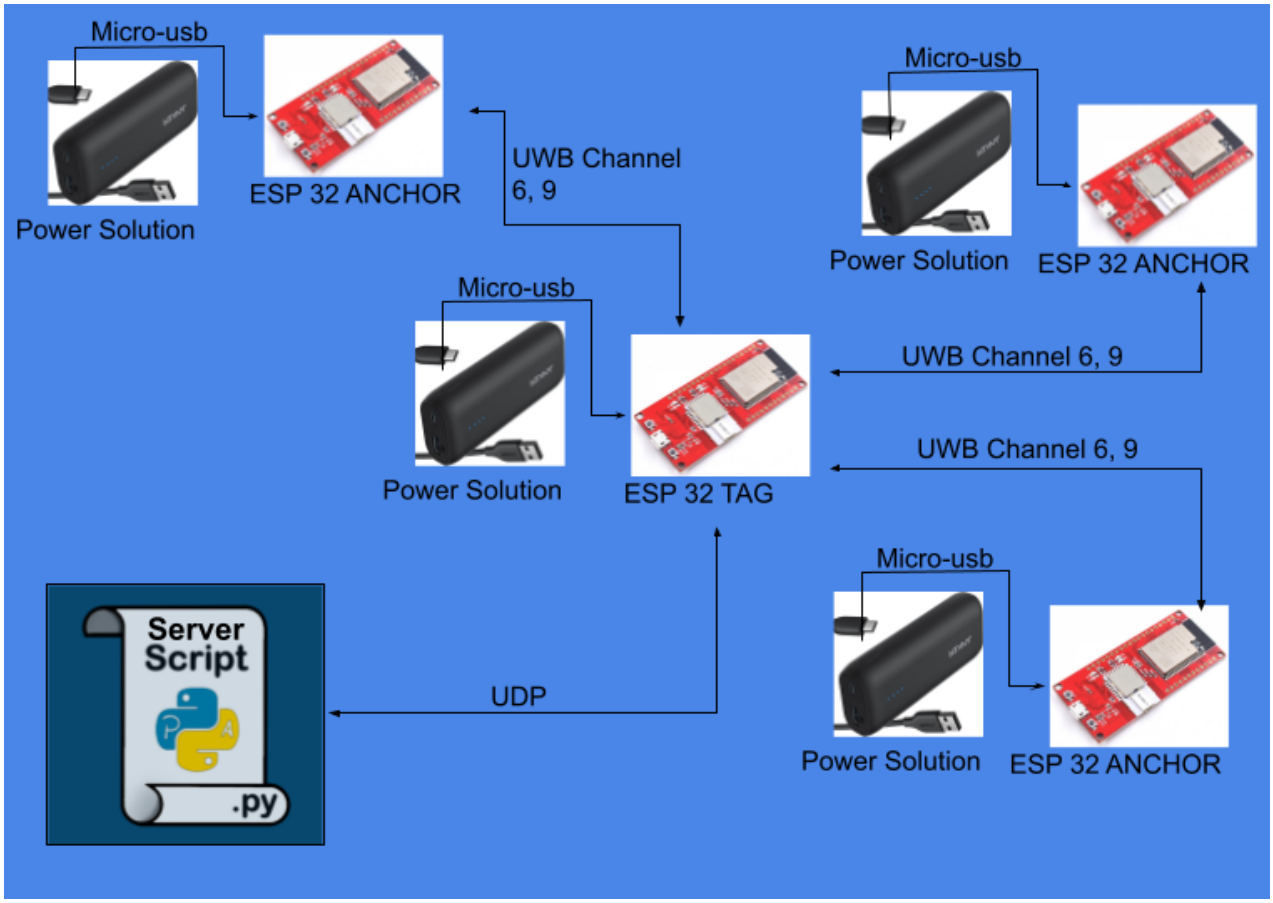


Figure 1: System Components & Connections

For our system testing, we primarily used the power bank shown, or similar, to power every device. However, in actual use, the power solutions can differ, as shown in Figure 2:

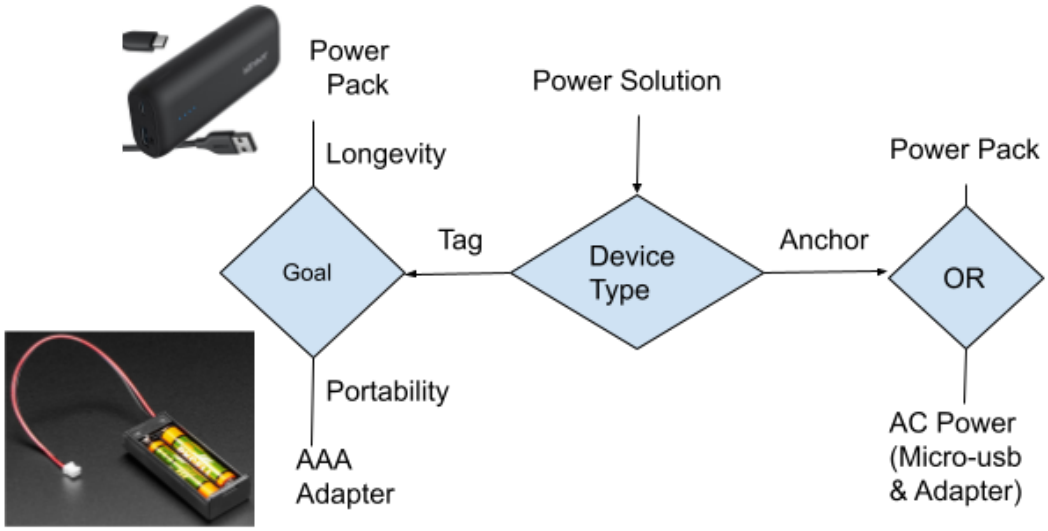


Figure 2: Power Selection

The main foundation of our system is the ESP32 UWB DWM3000 board. The entire board, by usb, is rated for 4.8~5.5V, 5.0V typical voltage. On this board we chose the WROVER over the WROOM microcontroller unit(MCU) available for this board. This MCU is slightly more expensive, 16MB more PSRAM memory, & other memory improvements. This DWM3000 is the successor to the UWB chip, the DWM1000. The DWM3000 can be used in 2-way ranging, TDoA and PDoA systems to locate assets to an accuracy of 10 cm(per its data sheet). The DWM3000 is backwards compatible with the DWM1000 & beats the DWM1000 by being 3x as power efficient. All together, this board supports wifi, bluetooth,

arduino, beta-access to Apple's U1 chip, & advanced UWB technology. During active signaling, the board uses these consumption ratings:

Mode	Power Consumption
Wi-Fi Tx packet 13dBm~21dBm	160~260mA
Wi-Fi/BT Tx packet 0dBm	120mA
Wi-Fi/BT Rx and listening	80~90mA

Figure 3: ESP32 Power Consumption

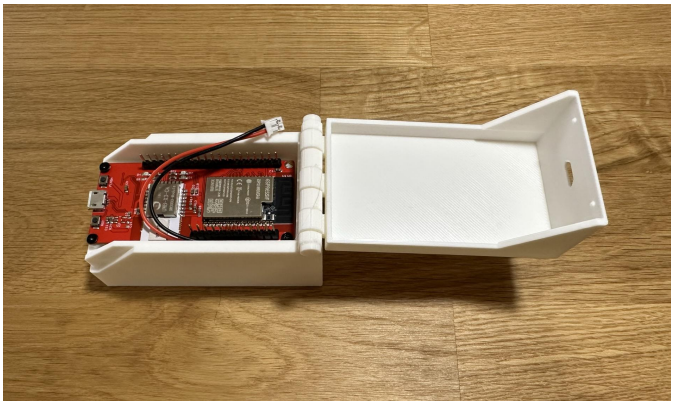
The power pack we most often use has a 5,200mAh battery & up to 12W of maximum power. This power pack provides vastly more life to the tag device, compared to the other power solutions we very briefly tested. We never had to charge our power pack through hours of testing.

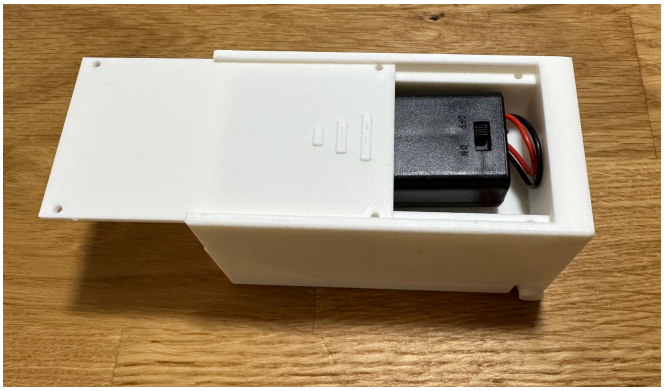
Our other power solution is the AAA battery adapter. The adapter we worked with briefly connected 2 AAA batteries & yielded about 3V.

Shell

The final version of our shell is an assembly of 5 pieces totaling a size of roughly 45mm wide, 85mm deep, & 45mm tall. We designed our shell to safely hold our ESP tag device & its power solution in one convenient easy place. So we started by designing it around the board, using the dimensions from the supplied pcb file & the listed dimensions for the power solutions. Then, to accommodate all possible solutions, we also placed 2 holes in between the board & power. For these holes, we had to go through several revisions in shape & size. In order to print without standoffs, so as to minimize material & print time, we needed to utilize angled lines. This started as an equilateral triangle, however, the shape could not allow the battery adapter through without being too large. So we switched to a pentagon shape, that ultimately was 1.5x the length of the triangle we'd hoped to use. After our first prototype, we realized a need for a more accessible cover. After going through ideas, we decided on a separate cover held in place by a hinge-like design & locking mechanism. Thankfully, the hinge was able to be cut into the shell, so we did not have to add any extra height. And it was made to be roughly equal sections across, to be sure the cover had ample area to keep together. Then we also made a custom pin to make our hinge function properly. For locking the pin, we implemented a thread inside one end & made our own screw cap with the corresponding thread to screw into the pin. Then to lock the cover, we had very little room to work with as we were

nearly inside the board & battery space. So, we utilized a nub & indent duo to keep the cover down. By placing spherical nubs on the front inside of the cover, with a spherical indent in the corresponding location, we kept the cover in place with a minimal footprint. Then we cut a hole in the covers front, that could fit most micro-usb cables, to be able to access our ESP device. Finally, we made a cover for our power solution. We initially designed it following the cover of the AAA battery adapter, but that was problematic. So we ended with a flat cover that goes through a rail the length of the shell, held in place with the same nub & indent duo on each rail.





Software Design

Functionality

The functionality of our system can be divided into four main systems These are the antenna delay calibration system, positioning system, automatic anchor calibration, and the positioning server.

Antenna Delay Calibration

As UWB radios measure range using time of flight, microseconds in delay can result in hundreds of meters of error. The antenna of the UWB radio introduces some delay. To solve this, we need to determine how much delay there is when sending and receiving data. This calibration only needs to be done on one device in a pair, but it needs to be done for every pair you intend to range with. The optimal delay is calculated through a binary search, where the delay is initially set to a number very far away from the optimal value, and the number gets smaller or larger depending on the difference between measured distance and actual distance. If the new value is found to be closer, then the delta for the delay is halved. This delta is halved until it reaches a number close enough to the actual antenna delay value.

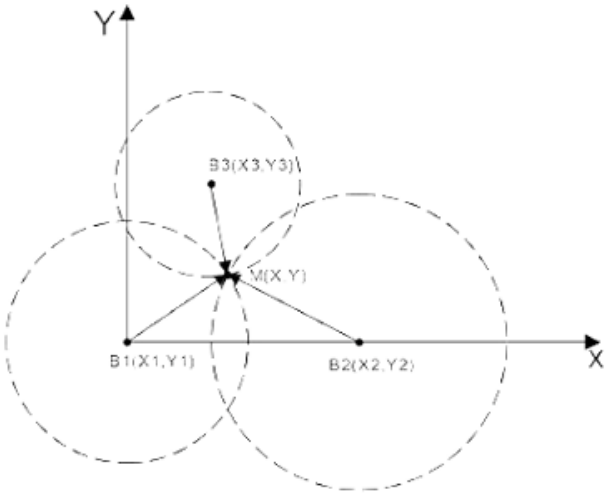
We originally thought that this code would be provided to us by the manufacturer, but as the DW3000 system is relatively new, a solution for this

specific radio had not yet been developed. In order to create our solution, we took a look at how the solution for the DW1000 worked. We took parts of that system, and integrated it to work with our devices.

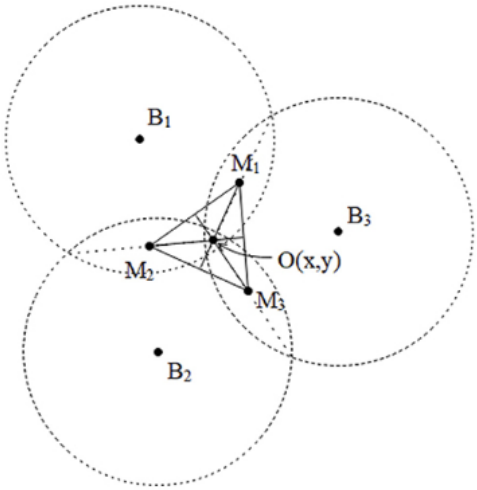
Positioning System

In order to calculate the position of the tag, the system needs to first determine the range from all three anchors. Using the ultra wideband radios, the range can be found by first sending a message to the anchor, which sends a message back, then calculating the time it takes to travel. This travel time is called time of flight, and it can be multiplied by the speed of light to calculate the range.

In our testing, we have found that the ESP32 Ultrawideband module is able to determine the range within a four-inch accuracy. These ranges are then used to triangulate the position of the anchor, and the position where the three radii intersect is the coordinate where the tag is, as shown in the figure below



This method is flawed however, as in most cases, the radii don't perfectly align like in the top example. Because of this, we needed a better way to determine the position. We found that the ideal way to determine position is to use the center of the intersection line between the circles to create a triangle, then finding the center of that triangle to determine the position, as shown in the figure below.



To implement this functionality, we found that there is no existing implementations of this using the DW3000 device we had on hand. Because of this, we needed to modify some existing implementations.

First, we needed a way to determine tags from all three anchors. The library we used had an example that showed us how to determine the range of a specified device using a mac address that we can define in the software. We adjusted this code to allow multiple devices to connect to the tag. This was the first technical

challenge we faced, which we will discuss later in the technical challenge section of the report. We then found code online that does the calculations for the positioning, and we adapted the code to work in our system.

Automatic Anchor Position Calibration

For the positioning system to work, the coordinates of the anchors need to be known. This can be done by physically measuring out the distances between the anchors, but this is time consuming, and not user-friendly. An alternative solution is to use the ranging capability of the UWB radios, and measure out the distances between all the anchors. To determine the coordinates of the anchors from the range between the anchors, we can do the following.

First, we set anchor A to the coordinates 0,0. Next, we set the x position of anchor B to be the distance between anchors A and B, and set the y position to 0. Lastly, to determine the location of the third anchor, we will use the law of cosines to determine the x and y coordinates, as shown in the equations below

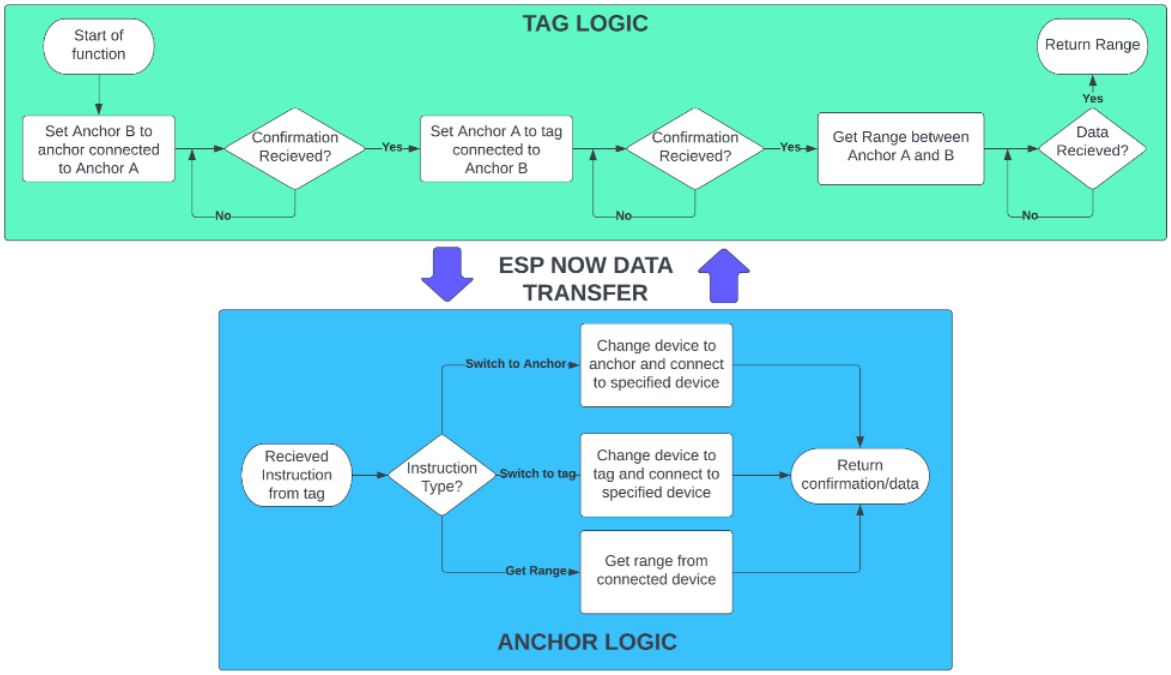
$$cos(\alpha) = \frac{b^2+c^2-a^2}{2bc}$$

$$x = b \cdot cos(\alpha)$$

$$y = b \cdot \sqrt{1 - cos^2(\alpha)}$$

Calculating the anchor coordinates is the easy part however, as long as you have the ranges between anchors. To determine the distance between the anchors, the following need to be implemented. First, the anchors need the ability to switch between anchor and tag modes Next, we need the measured distances to be sent to a central device. Lastly, the central device needs to be able to communicate with all of the anchors to synchronize the process of switching modes, and sending instructions to measure the range.

To solve these issues, we adjusted the code for the anchors and tag as follows. First, we added snippets of code from the tag to the anchor to allow the device to switch between modes. Next, we used the ESP now protocol to communicate between devices. ESP now is a protocol that uses the wifi antenna to communicate between esp devices. Lastly, the central device will use is the tag, and it will send instructions to the anchors that will allow us to read the distances between them.



This flowchart shows how the function to get the ranges between anchors works. For each instruction sent to an anchor, the tag will wait for a confirmation message to be sent from the anchor, to ensure that the instruction is completed successfully and fully. This is required, as some instructions are reliant on previous instructions to be executed fully to work, such as getting the range in the final step. In both the anchor and the tag, we created message parsers that edit global variables that control what the devices do. The reason we needed to use global variables is that the ESP Now communication runs on a separate processor, so using global variables, we are both able to set up synchronization between tasks in the processors, and communicate between the processes.

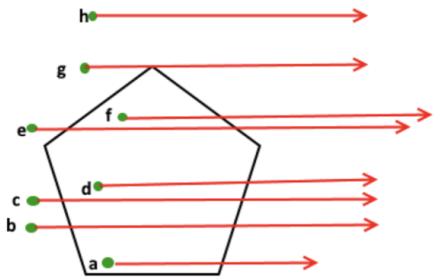
On the anchor, the message parser converts the instructions into tasks that the device completes. The message parser will then send a confirmation message back to the tag that includes completion status. If the instruction was a ranging instruction, it will send the range data back along with a confirmation that lets the tag know if the data is valid or not. Back on the tag, the message parser determines completion status, and sets a global variable to the recieved completion status. If it is sent data, it will also save the data into a different global variable.

The code required to implement this system was completely custom, and was the most difficult part of the project to complete. The parts we needed to implement included, implementing ESP Now into the system, creating a messaging system and message parsers for both the tag and the anchors, and adding the ability to switch between anchor and tag modes on the anchors.

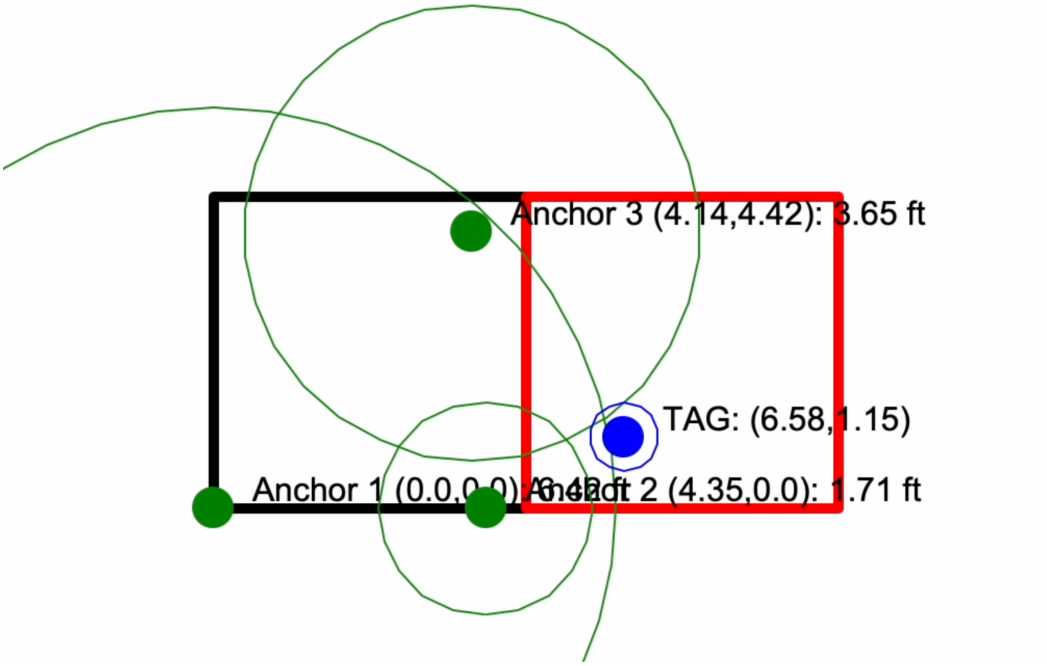
Positioning Server

There are two parts of this server notable to talk about. The first part is the room definition system. This system comprises a series of menus that walks the user through the process of adding a room. To get the coordinates of each corner of the room, the system takes an average of 30 measurements. The data is stored in a room object that contains a polygon object, the name of the room, and the room ID. The polygon object holds a list of point objects, which holds the x and y coordinates of each point.

The next notable section is the actual positioning server and graphical display. This server receives the position of the tag, the anchors, the ranges from each anchor, and the calculated error in tag position, and uses it to draw out all of the positions. It also draws out the rooms defined in the previous section, and draws the currently inhabited room in red. It determines what room the user is in using a system that determines whether a point is within a polygon, explained in the figure below.



This involves drawing a horizontal line starting at the point, going to the left, and determining the number of edges in the polygon that the line intersects. If the number of intersections is odd, then it is inside the polygon.



The figure above shows what the final system looks like. The room the tag is in is displayed as red, and the ranges from each anchors are displayed with the green circles around the anchor. The blue circle around the tag shows the possible area that the tag can be inside, determined by the error calculated when determining the position.

The code for the positioning server was based off of a similar project created by the people who designed the ESP32 board we use for the project. The project was created for the DW1000 version of the board however, and was much more limited in features. We based the positioning server code off of their positioning server, but most of the code for the server is custom. The main parts of the code we took from was learning how to draw graphics using the python turtle package.

Security

When developing our system, security was one of our main concerns. As our system is able to determine the precise location of its users within a household, the data that is gathered is very sensitive. Because of this, the safety and security of our system is a big focus. In our system, messages are sent between the ESP32 devices to determine the range. To ensure security between the device's communication, the messages sent between the ultrawideband radios of the ESP32 devices are encrypted using AES. This encryption ensures that the messages that are received by the device are being sent from the expected device. This reduces the chance of the system being compromised by rouge devices acting as devices in the system. When messages are received, it is compared against the expected message stored in each device to ensure that the correct message was sent.

Programming Languages Used

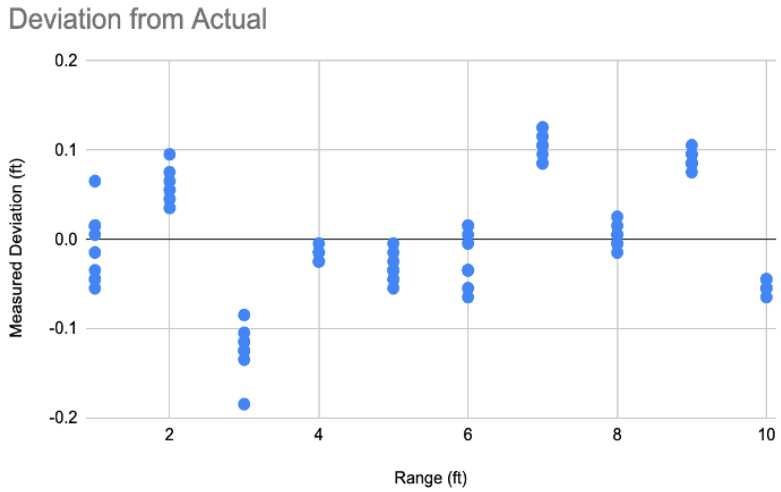
This project uses two programming languages. The code for the ESP32 devices are programmed in C++ using Arduino IDE to upload the code, and the code for the positioning server is coded in python. We went with C++ for the ESP32 as the libraries used are programmed in C++, and we used Arduino IDE to upload the code. We coded the positioning server in Python as it is has great tools such as the python turtle package that allow us to implement the features we wanted.

Related Work

We were only able to find one project that is similar to what we did. This project was created by the people who designed the ESP32 DW1000 module, and only works on that module. Their system involved using two anchors and one tag, so it was not a true positioning system. The method they used of obtaining the position of the tag is identical to the method we used to determine the position of our third anchor in our automatic anchor position calibration system, which is using the law of cosines to determine the position. This project was what gave us the idea to implement the automatic anchor positioning system. As there are only two anchors, it can only determine the position on the positive y-axis, where the anchors lie on the x-axis. This system can only provide the absolute y value, so if you are at (2,-3) it will read as (2,3). This project is also what we took from to start our positioning server, as the project showed us how to use Python Turtle graphics. Our positioning server code is mostly custom, but was originally based off of their code.

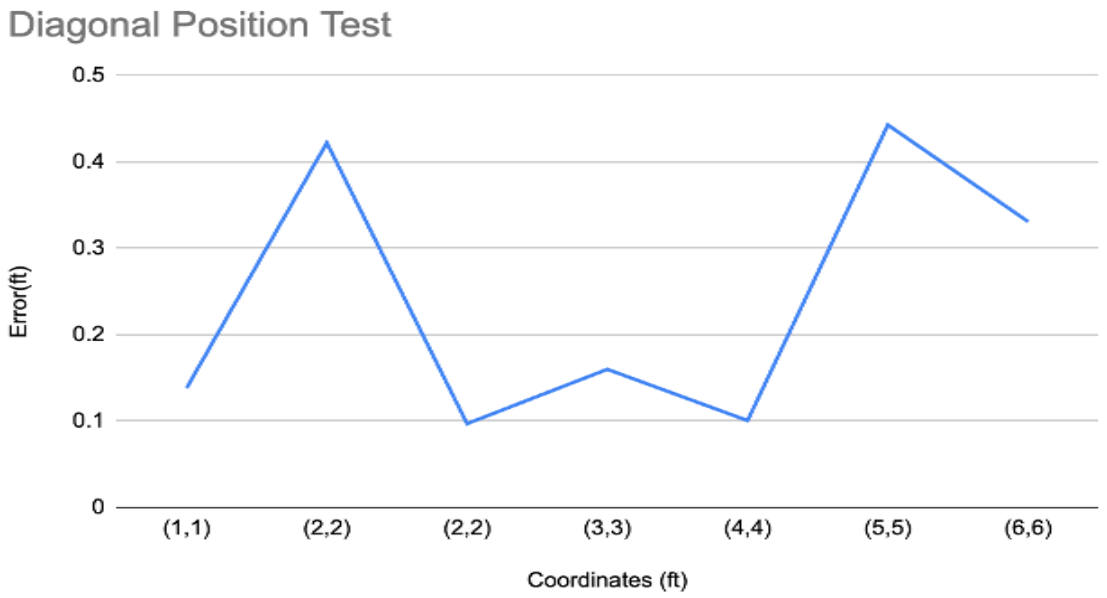
Results & Discussion

When we started this project, we did some initial testing to evaluate the accuracy and achievability of implementing our system within our design constraints. The first test we conducted was a range test.



The above scatterplot shows the results of our ranging test from foot-long intervals between one and ten feet. For each interval, we took 10 measurements. The data shown on the graph show the deviation from the actual distance. To gather the data, we created custom code to gather the data. We modified the positioning system to instead take the range, and input them into a csv file. Creating this testing system sped up our evaluation process by automating much of the manual tasks we would have needed to do. This test was done by placing two tables, one for the tag, and the other for the anchor, at foot long interval distances from each other. The floor tiles in the lab were found

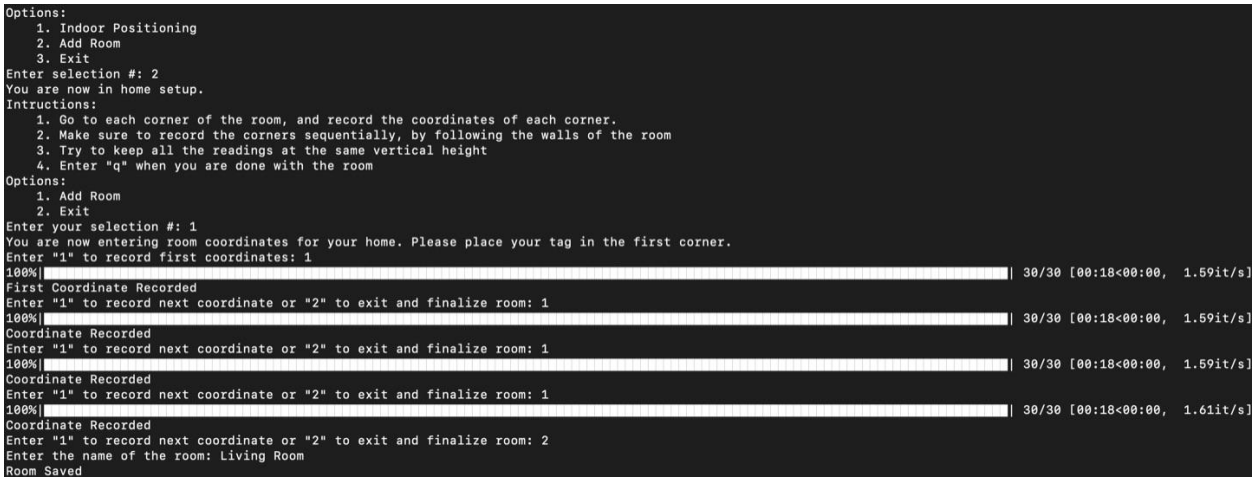
to be exactly one feet each, so we moved the table with the anchor from tile to tile. The results show that the accuracy is generally within four inches of each other. You can also notice that the groupings are very tight for each interval, pointing to an error in testing. This is likely caused by errors in the actual distance when we conducted our test. If we look at the range of values for each interval, the ranging accuracy becomes two inches. This is well within our design constraints, and will allow for a system that has a final accuracy of within six inches of the actual position.



Next, after we completed our initial positioning system, we conducted a positioning test. This test was conducted at one foot diagonal positions, from the position (1,1) to (6,6). As you can see from the data, all of the measurements are within half a foot of the actual position, showing that our system can work within our design constraints.

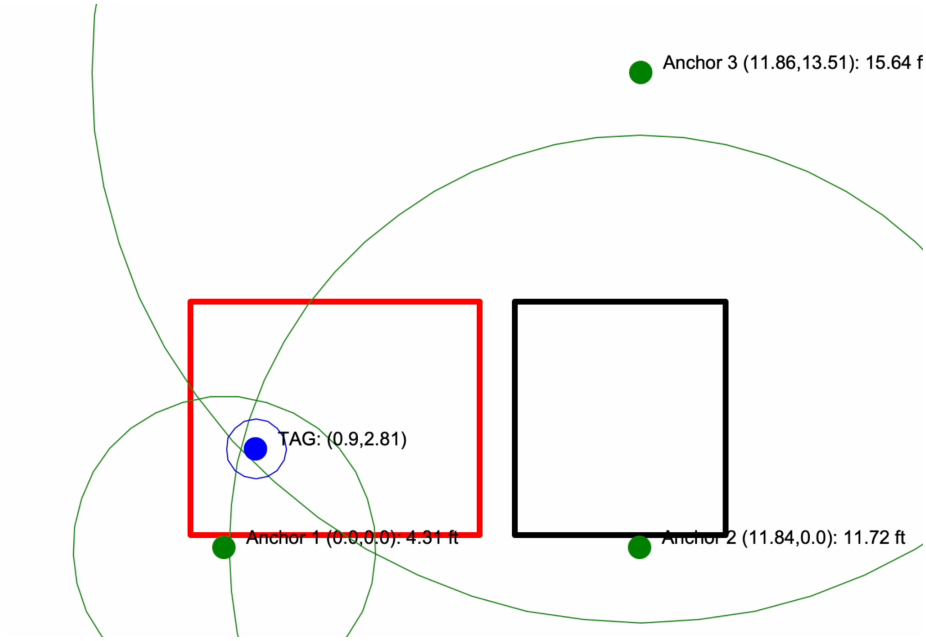
Final System

Home Initialization



The screenshot above shows the process of adding a room to the system. The console guides you through the process step by step, with instructions at the beginning to guide you on the methods to use to get the best results. For each coordinate entered, the system takes 30 measurements. This allows the system to average out the readings to get the highest accuracy.

Positioning System and Graphics



The Figure above shows what the finalized positioning system looks like. As you can see, the rooms that the user defines are shown on the display as the rectangles. The room that the user is currently in is highlighted in red, and the other room is colored in black. This system displays live positional data, and updates every second. In real world applications, the system would not need to be constantly polling for location, and would only need to measure the position when a request was made, for instance if the user asks the voice assistant to turn on the lights, it will poll for position to determine the room the request was made from, and turn on the lights in that room. This would also drastically reduce energy consumption compared to the amount of energy the current implementation runs.

Technical Challenges

Case Design Challenges

The main challenge was getting the pieces to fit together smoothly. Early on we discovered that the 3D prints would come out slightly larger than the 3D models, about 0.2mm. This was present in the initial cover & locking mechanism, & the battery cover for several iterations. Which resulted in pieces that didn't fit at all as intended, or sometimes just a bit offset from ideal.

Another challenge was standoffs for the ESP board. With little space to work with, we had to find the right balance between size & strength. Our first few standoffs did not hold during printing, or collapsed when we tried to screw into them.

The final challenge we had was the wire holes for the battery adapter. For these holes, we had to balance shape, size, & orientation. We needed to make holes big enough for the pin piece to get through, but not too big to where we compromise the board's floor. And, given the general location would be almost perfectly in the middle of the board, we had to utilize angles in order to prevent standoffs during printing. Which meant a smaller overall hole, compared to a circle or square.

Connecting Multiple Anchors

One of the first challenges we faced when implementing the positioning system was the ability for the tag to connect to multiple anchors. The issue was that we were able to get the tag to connect to each anchor individually, but when trying to get the ranges from multiple anchors at once, the anchors would stop responding.

We found out that this issue was due to several factors. First, we needed to create an object for each device inside the tag code. This object will include data such as the anchor's address, the AES keys and config, and a lot of data about the current state of the DW3000 device. This fixed one of the issues we were having, and made it much simpler to switch between devices in the code.

The main issue we ran into was the way that the anchors handled errors. The sample code included in the DW3000 library would trigger an infinite while loop when the device receives a message that was not intended for it. This is an issue, as the tag sends messages to all three anchors. To fix this issue, we made the function return to the main loop when it receives a message not intended for it, rather than get stuck in the loop.

Implementation of Automatic Anchor Position Calibration

We ran into a wide variety of issues when implementing the automatic anchor position calibration. The first issue we ran into was getting ESP Now to work at

the same time as WiFi. As the tag needs to both send and receive data to the anchors over ESP Now, and send data to the position server over WiFi UDP, we need both protocols to be able to run on the same device. The issue we ran into was that when WiFi capabilities were added, ESP Now would cease to function. We were able to get ESP Now to work in a simple test with no other functionality, but as soon as we added it to our main code, it stopped working. We found that the issue was a result of the ESP32 only having one 2.4 GHz radio, which is used for both the WiFi and ESP Now protocols. To resolve this issue, we created functions that would turn off one of the standards and initialize the others. The functions we created are shown below.

```
int switch_esp_now() { // Switch from WiFi to ESP NOW
 WiFi.disconnect(true);
 WiFi.mode(WIFI_OFF);
 WiFi.mode(WIFI_STA);
 if (esp_now_init() != ESP_OK) {
   Serial.println("Error initializing ESP-NOW");
   return -1;
 }
 esp_now_register_send_cb(OnDataSent);
 if (esp_now_add_peer(&peerInfo1) != ESP_OK) {
   Serial.println("Failed to add peer 1");
   return -1;
 }
 if (esp_now_add_peer(&peerInfo2) != ESP_OK) {
   Serial.println("Failed to add peer 2");
   return -1;
 }
 if (esp_now_add_peer(&peerInfo3) != ESP_OK) {
   Serial.println("Failed to add peer 3");
   return -1;
 }
 esp_now_register_recv_cb(OnDataRecv);
 delay(5);
 return 0;
}
```

```
void switch_wifi() { // switch from ESP NOW to WiFi
 esp_now_deinit();
 connectToWiFi(ssid, password);
 WiFi.mode(WIFI_AP_STA);
}
```

These functions solved the issues we had, but came at a cost. Initializing WiFi takes several seconds to complete, and would stall the program. To resolve this, we designed our automatic anchor position calibration process so that this switch only needed to happen once during the whole process. Since the anchor calibration process only needs to happen once, the computational cost of running this switch once is insignificant.

The next issue we ran into was switching between anchor and tag modes on the anchors. To do this, we copied large parts of the code from the tag and placed it into the anchors. We also updated the device class to include extra data needed to run both modes. Lastly, we created functions that switch between the modes, which initialize the parts of the DW3000 antenna needed to run either mode. This solution worked well in isolation, when the mode switches were hard coded into the anchor, but when getting the instruction to switch modes from the tag, it would stop functioning entirely. We spent many hours trying to figure out this issue, including a complete rewrite of the anchor code, but nothing seemed to solve this issue. We eventually figured out that the issue was a result of the issues with running the DW3000 in a multithreaded application. As the ESP Now protocol functions in a separate core so that it is able to receive messages on demand, the program is required to run on multiple cores. This caused issues

with reading data from the status register on the DW3000. When the status register fails to read, a section of the code that detects errors is triggered and the program stops. To resolve this issue, we did a lot of research, and found a function in the DW3000 library that does a soft reset of the DW3000 radio. This cleared up any issues with the status register that resulted from switching modes in the multithreaded program, and allowed the program to function properly.

User Manual

In this user manual, we will walk you through how to initialize the system, and then how to operate the system.

Initial System Setup

Antenna Delay Calibration

If you are setting up the system for the first time with new devices, you will need to determine the antenna delays for the following combinations of devices.

- Tag to Anchor 1
- Tag to Anchor 2
- Tag to Anchor 3
- Anchor 1 to Anchor 2
- Anchor 1 to Anchor 3
- Anchor 2 to Anchor 3

To calculate the antenna delays, you will place your anchor device and your tag at a measured distance. This distance will then be entered into the software global variable defined at the top of the sketch called “this_anchor_target_distance”. Once you have this set up, upload the anchor and tag codes to the respective devices. Open the serial monitor, and you will see the calculated anchor delays for the pair.

Once you have determined all of the anchor antenna delays, you will need to enter them into the sketch for the anchors in the indoor positioning file. You will see a section near the top of the file beginning with the #if statement “#if anchornum == 1”. For each statement, that statement will contain the unique data that each anchor will contain. Inside of these statements, you will see several antenna delay definitions. For each pair of TX/RX delays, keep the value the same. The first delay you will see is for the tag. Enter the antenna delays you got between the tags and the anchors for each respective anchor. Next you will see the antenna delays between the anchors. Enter the antenna delays you determined earlier for each anchor. You will also see that for each anchor, the antenna delay for that anchor is also a value, for instance, the definitions for anchor 1 will contain the antenna delays for anchor 1 as well. For these values, the default value of 16384 will already be entered, so you do not need to change these values.

Tag Initialization

In the code for the tag, you will need to enter in your wifi credentials, including ssid and password. You can find these variables near the top of the file. You will also need to enter in the ip address of the host computer that the positioning server will run on. Once these are entered, you can upload the code onto the board.

Uploading anchor code to the ESP32

To upload the anchor code onto the ESP32, all you will need to do is specify which anchor you are uploading to, defined in the top of the file in the macro called anchornum. This ranges from 1 to 3, for the three anchors you will be using for the system. Change this number for each anchor you upload the code to.

Setting up Positioning Server

In the positioning server file, there is only one value that needs to be adjusted. This variable is located near the top of the file, and is called UDP_IP. Enter in the local ip address of the system you are running the server on. This address should be the same address you entered into the tag.

How to use the system

System Setup

To set up the system, power on all three anchors and place them a good distance apart from each other. Try to keep the anchors at the same height. If you need to place the anchors at different heights, you can measure out the heights and input them into the tag code inside the variable called anchor matrix. The Z height is the third variable on each line. A Z height of 0 is defined as the height you expect the tag will be most of the time. For instance, if the tag

is to go in your pocket, that height will be defined as 0. If the anchor is placed on a table 1 foot higher than your pocket, then the z height will be defined as 1.

Once you have determined the heights, you can now power on the tag. When the tag powers on, it will initialize the anchor positions using the automatic anchor calibration process we created.

Defining Rooms

With all three anchors and the tag powered up and ready, you can now start the positioning server by running the python file. On startup, you will be directed to the main menu. To define rooms, you will select option 2, add room. To select the option, enter the number 2 into the console. You will then be directed into the room setup. Read the instructions printed out, and enter 1 to add a room when you are done. Next, you will be entering the coordinates for the room. To do this, bring the tag to the first corner, and try to keep the height of the tag at the height you defined to be Z = 0. Next, enter 1 into the console to start the distance measurements. You will now see a progress bar detailing the progress of the measurements. Once that is complete, move to the next corner. It is important to move to each corner sequentially by following the walls. Do not cut corners. Enter 1 again to record the coordinates. Repeat these steps until you have recorded every corner of the room. Make sure not to repeat any corners. When you are done, enter 2 to finalize the room. You will now be instructed to enter the name of the room. When you finish the room, you can either choose to

enter another room, or exit to the main menu. Exit to the main menu when you have completed entering all of the rooms.

Start Positioning

Next, you can start the live positioning. To do this, enter 1 into the main menu. You will now see the rooms you defined in the previous step, as well as the position of your tag, and all three anchors. As you walk through the rooms, the room you are currently in will be outlined in red. You can now use the system as you wish.

Team Member Contributions

Jake Black

- Designed and refined the case for the ESP32 over several prototypes
- Researched antenna delay calibration process and found an example implemented with DW1000 library.
- Researched the positioning system and learned how 2D triangulation works
- Helped debug several software issues relating to connecting multiple anchors to one tag.
- Conducted extensive testing with ESP32 and other devices

Peter Xiong

- Implemented antenna delay calibration process by converting DW1000 example found by Jake to use the DW3000 library
- Implemented positioning system using three anchors and one tag using research done by Jake as reference
- Designed and implemented automatic anchor position calibration
- Implemented positioning server
- Implemented room initialization and detection
- Conducted extensive testing with ESP32 and other devices

Conclusion

In conclusion, we achieved our goal of creating an indoor positioning system that is able to determine the precise location of users inside of a household. Our system also achieved our design constraints of being accurate to within half a foot. We also achieved the goal of adding the ability to define the dimensions of the rooms, and the ability to determine which room the user is in. Lastly, we successfully designed and created a 3D printed shell for our ESP32 device, that houses both the ESP32 module, and the battery needed to run it. We are quite happy with the results that we achieved throughout the duration of this semester.

Future Work

While we are quite pleased with our final result, especially given the number of challenges we ran into along the way, there are many things we would have liked to implement if given enough time.

We wanted to do more to calibrate our system, by introducing statistics to filter out the bad data gathered from the ranges. This would require extensive testing to determine how best to filter the results, which we did not have time to do.

We had also originally planned to create our own smart devices that can be used to demonstrate the functionality of our system, such as a simple arduino program that turns a light on and off depending on if the user is inside of a room. We also wanted to implement a smart home app that displays the rooms by distance in ascending order. This would have shown one of the largest benefits of our system, which is to simplify the home app experience by showing the most relevant devices at the top.

While we did run out of time, we found other ways to demonstrate our system, by displaying it using turtle graphics. While it is not the same as showing the results in real life with our own custom smart devices, it still was able to show off the functionality of our system.

References

- “Smart Home Market Size, Share, Growth and Trends Report 2030.” *Smart Home Market Size, Share, Growth & Trends Report 2030*, https://www.grandviewresearch.com/industry-analysis/smart-homes-industry#:~:text=Report%20Overview,devices%2C%20dishwashers%2C%20and%20more
- “Ultra-Wideband (UWB) Here's everything you need to know.” *Bleesk,* https://bleesk.com/uwb.html#:~:text=Ultra%2Dwideband%20(also%20known%20as,more%20precise%2C%20reliable%20and%20effective
- Bakr, M.“Introduction to ultra-wideband (UWB) technology - technical articles. All About Circuits, 2020, March 20. https://www.allaboutcircuits.com/technical-articles/introduction-to-ultra-wideband-uwb-technology/
- Editorial Team. “What is Ultra-Wide Band (UWB) Technology?” *everything rf,* 2021, July 15. https://www.everythingrf.com/community/what-is-ultra-wide-band-uwb-technology
- Gautier Bonnemaison. “Pose recognition system on Home Assistant and Drone Tracking with IR-UWB”, Dr. Saniie, 8/13/2021 ECASP, IIT
- Stephen Cooper, “Ultra-wideband & you”, WLPC Phoenix 2020 https://www.youtube.com/watch?v=TR-rahy3Y2k
- DWM3000 reference, datasheets & more: https://www.qorvo.com/products/p/DW3110#documents
- Project using DW1000 for Indoor Positioning https://how2electronics.com/esp32-dw1000-uwb-indoor-location-positioning-system/
- Reference used to learn how to detect if a point is inside a polygon. https://www.geeksforgeeks.org/how-to-check-if-a-given-point-lies-inside-a-polygon/

Appendix

1. WROOM vs WROVER

	ESP32-WROOM-32	ESP32-WROVER-B (16MB)
Description	Wi-Fi+BT+BLE Module	Wi-Fi+BT+BLE Module
CPU	ESP32-D0WDQ6	ESP32-D0WDQ6
CPU clock frequency	80MHz~240MHz	80MHz~240MHz
Operating voltage/Power supply	2.3 V ~ 3.6 V	2.3 V ~ 3.6 V
Operating current	Average: 80 mA	Average: 80 mA
Minimum current	500 mA	500 mA
Crystal oscillator	40MHz	40MHz
Radio frequency	2412~2484MHz	2412~2484MHz
SRAM	520KB	520KB
GPIO	26	24
GPIO for Flash/PSRAM	6,7,8,9,10,11/	6,7,8,9,10,11/16,17
PSRAM	No	8 MB
Flash	4 MB	16 MB
Peripherals	capacitive touch sensors, Hall sensors, SD card interface, Ethernet, high-speed SPI, UART, I²S and I²C, SDIO, LED PWM, Motor PWM,IR, pulse counter, GPIO,ADC, DAC	capacitive touch sensors, Hall sensors, SD card interface, Ethernet, high-speed SPI, UART, I²S and I²C, SDIO, LED PWM, Motor PWM,IR, pulse counter, GPIO,ADC, DAC
Operating temperature range	–40 °C ~ 85 °C	–40 °C ~ 85 °C
Size	(18.00±0.10)mm×(25.50±0.10)mm×(3.10±0.10)mm	(18.00±0.10)mm×(31.40±0.10)mm×(3.30±0.10)mm

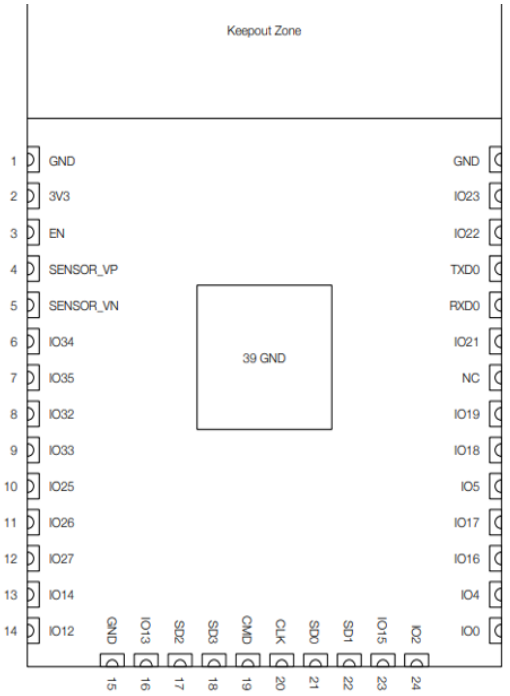


Figure 1: ESP32-WROOM-32 Pin Layout (Top View)

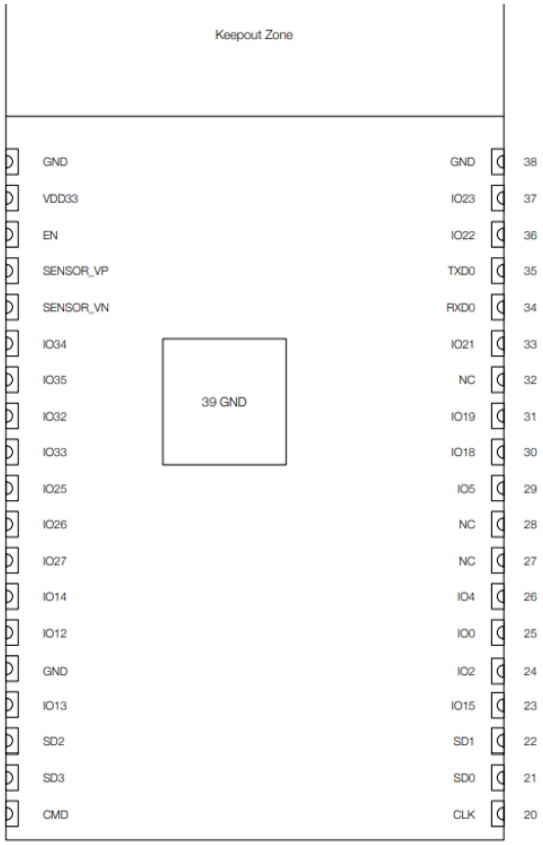


Figure 1: Pin Layout of ESP32-WROVER (Top View)

2. Bill of Materials

Name/Link	Description	Price	#	Total
ESP32-UWB-DW3000-WROVER Board	Electronic module that has UWB and other radio protocol capabilities. Arduino Compatible 3.3 and 5V power source capable. Board USB supply voltage range: 4.8~5.5V	$45.80	4	$183.20
3D Printed Shell	Shell for protecting ESP boards & holding Tags power solution	$0.32	4	$1.28
PowerB: AnkerPower Bank	Quality, reliable power pack that is similar in size and shape to ESP module. Includes USB-A to C cable 5,200mAh, 5V, 12W	$24	1	$24
ConnectorB: Micro-usb cable	Micro-USB to USB-C cable to connect board to power pack. 2.4Amp, up to 480 Mbps	$8	1+	$8+
AAA Battery x4	Portability-focused power option for tag	$8	1	$8
AAA Battery Adapter x3	Adapter for converting AAA batteries into a plug that can fit ESP pins	$7	1	$7
Total				$231.48