

Parking Reservation System

Team 59

Manjesh Mogallapalli, Ojus Deshmukh, Vivek Calambur

TA: Kyle Michal

Final Report

ECE 445 - Spring 2019

May 1, 2019

Abstract	3
1 Introduction	4
1.1 Problem Statement	4
1.2 Solution	4
1.3 High Level Requirements	4
2 Hardware Design	5
2.1 System Overview	5
2.3 Power Supply Subsystem	6
2.4 Meter Subsystem	6
2.5 Microcontroller Code	9
3 Backend Design	11
3.1 Backend Overview	11
3.2 Firebase and Google Cloud	11
3.3 Wi-Fi Chip Code	12
3.3 Spot Assignment Algorithm	12
3.4 RFID Verification System	13
3.5 Reservation Enforcement System	14
4 Mobile App Design	15
4.1 Mobile App Overview	15
4.2 Mobile App Frontend Design	15
4.3 Mobile App Backend Design	16
5 Verification	17
5.1 Hardware Verification	17
5.2 Backend Verification	17
5.3 Mobile App Verification	17
6 Costs	18
6.1 Labor	18
6.2 Parts	18
7 Conclusion	19
7.1 Accomplishments	19
7.2 Challenges	19
7.3 Future Work	20
7.4 Ethical Considerations	20
7.5 Acknowledgements	20
8 References	21

Appendix A Requirements and Verification Table	22
Appendix B PCB Schematics	25
Appendix C PCB Layouts	27
Appendix D PCB Pictures	28
Appendix E Physical Design	30
Appendix F Mobile App Layout	32
Appendix G Schedule	36
Appendix H Project Data Flow	38

Abstract

With the advent of IOT and smart devices, many day-to-day devices now have the power to connect to the internet. While certain devices such as smart refrigerators and thermostats are easily found on the market today, parking meters have been largely unaffected by recent technology. The parking reservation system detailed in this paper brings parking into the modern world. By building parking meters that are capable of connecting to the internet, users will be able to view open parking spots, reserve spots and pay for parking with a simple tap. The final product consists of parking meters that can be sold to cities, private lots or garages, and a mobile app that users can download and use.

1 Introduction

1.1 Problem Statement

We have all been in situations where we find ourselves driving aimlessly looking for public parking or having to park out of the way because no spots are available. This problem gets worse as cities grow—drivers in London spend an average of almost eight minutes looking for parking every day[1]. In the US, after accounting for time, fuel, and emissions, drivers lose over \$70 billion[2] annually while looking for parking. Additionally, users often overpay for their parking spot. The average driver in the US spends “eight times more a year overpaying for parking than they do in parking tickets”[2]. These issues stem from not having a comprehensive system for parking that allows a user to effectively find available spots or to know that a spot will be saved for them on arrival.

1.2 Solution

Our solution is a comprehensive parking system that allows users to easily gauge the parking availability in a certain area before they start driving or simply reserve a spot ahead of time for a set window. Instead of having to deal with coin payment, garage systems, or blocked-access parking lots, the system will rely entirely on a parking meter and a companion app. Users will also receive an RFID tag they will use to check in to parking spots.

The system works as follows:

1. The user either reserves a spot, or finds an open spot on the mobile app
2. The user parks at that spot, and scans their RFID tag on the meter
3. When the user drives out of the spot, their account will immediately be charged for the time they were parked at that spot.

1.3 High Level Requirements

- ✓ The meter will be able to verify users via RFID tags.
- ✓ The hub unit will be able to support at least 10 active meters
- ✓ The system will identify empty parking spaces with 90% accuracy

2 Hardware Design

2.1 System Overview

The system involves an array of meters, connected to mains, that all communicate with a backend service. This backend service receives sensor data from the meter, and in turn communicates to the meter what to display on its LCD screen. The mobile app also communicates with the backend in order to obtain information on spot availability and to make reservations.

This data flow can be seen in Figure 1:

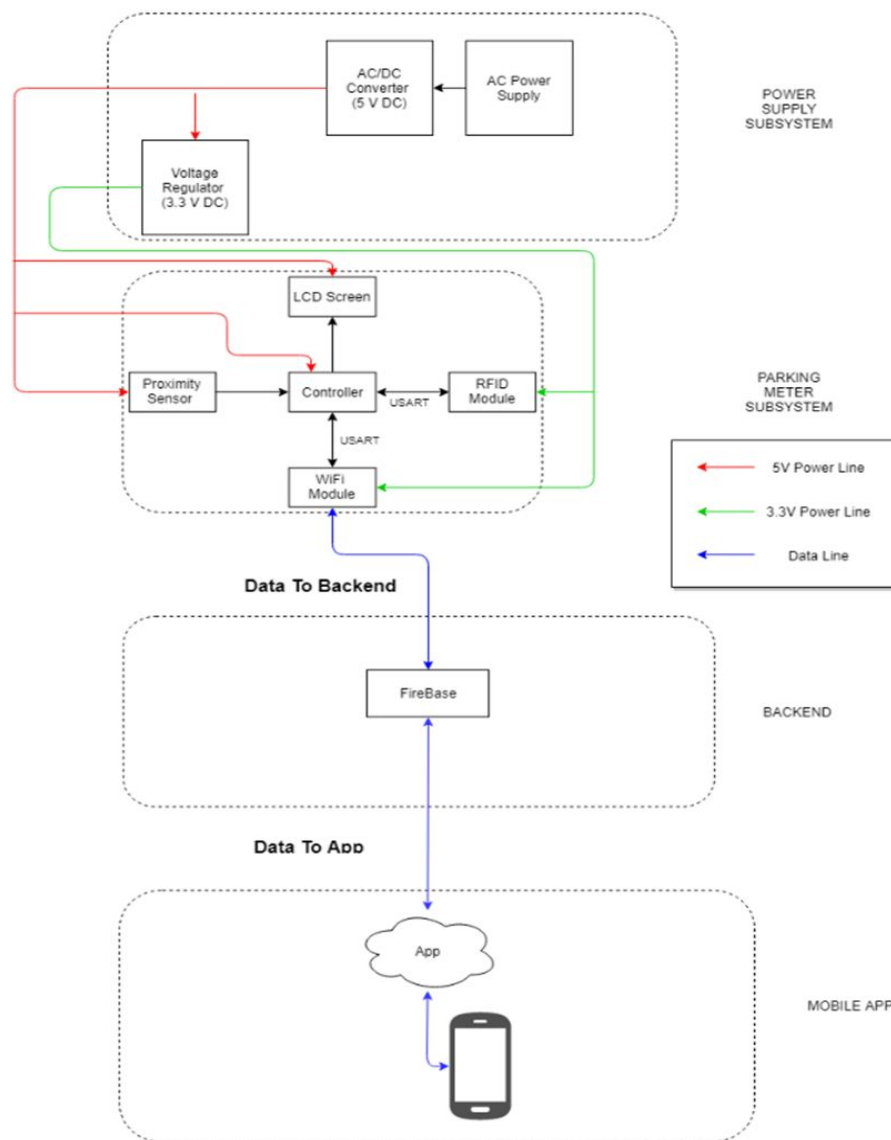


Figure 1: System-Level Block Diagram

The hardware for each meter unit consists of a power supply subsystem, and the meter itself. These meters can then be individually deployed at any parking spot to detect cars and communicate to the backend service.

2.3 Power Supply Subsystem

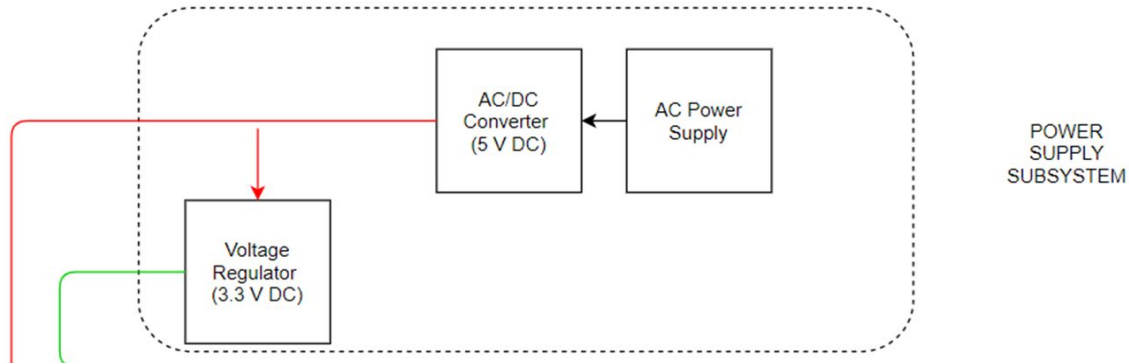


Figure 2: Power Supply Subsystem

Each power supply subsystem is responsible for powering its own meter, and ensuring that it is always connected and powered after the meter boots up. This system contains two components - an AC/DC 5V converter and a 3.3V linear regulator.

Converter: This is a 5V, DC power source that can supply up to 2A to its respective meter.

Regulator: This is a 3.3V, DC power source that supplies 3.3V power to the WiFi module onboard the meter.

2.4 Meter Subsystem

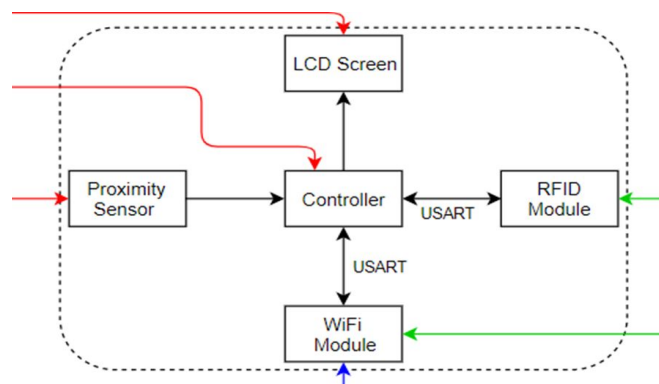


Figure 3: Parking Meter Subsystem

The meter subsystem contains 5 components:

1. An ATmega 328-P microcontroller, that controls and reads from all the other components and sensors onboard the meter. This microcontroller was chosen because of its usage in Arduino platforms, meaning that all Arduino libraries would be compatible with this microcontroller.
2. An ESP8266-01 WiFi Module, that connects the backend and the meter. The ESP8266 platform has multiple boards that have more GPIO pins, and that could theoretically control the entire meter on their own without the use of an ATmega. We chose the ESP-01 because it was designed as an add-on module to other microcontrollers such as the ATmega, thereby allowing us to utilize Arduino libraries. This microcontroller had the biggest current draw, with around 67 mA in steady state with bursts of up to 435 mA while it connected to WiFi (see Fig.4). This WiFi module also had a library that made it very easy to interface with our backend, the Firebase-ESP8266 Client library[3].
3. An HC-SR04 Ultrasonic Sensor, that functions as a proximity sensor to detect cars parked in the lot. Ultrasonic was cheap, reliable, and easy to implement, as well as allowing us to make many quick measurements to reduce error.
4. An ID-12LA RFID Reader, that allows users to check in to parking spaces. We chose RFID as a user identification method because it was reliable and very portable. We debated a mobile verification method, but also wanted our system to be entirely viable without the use of a mobile phone. This particular model of RFID reader also came with a built-in antenna, which simplified our circuit and boosted our read range.
5. An NHD-0216K3Z-FSRGB-FBW-V3 LCD Screen, that allows the meter to display information about the parking spot. We chose an RGB backlit screen because they would be easier to identify at a distance in larger parking lots. It required I2C communication, which we did using the Wire I2C library[4].

Component	Voltage	Current
Microcontroller(ATMega 328-P)	5V	7-15 mA
Ultrasonic Sensor(HC-SR04)	5V	15 mA
RFID Reader(ID-12LA)	5V	15 mA
WiFi Module(ESP8266-01)	3.3V	67-500 mA
LCD Screen + RGB Backlight(NHD-0216K3Z)	5 + 1.9,2.9,2.9 V	15 + 15,20,20 mA

Figure 4: Consumption Characteristics by Part

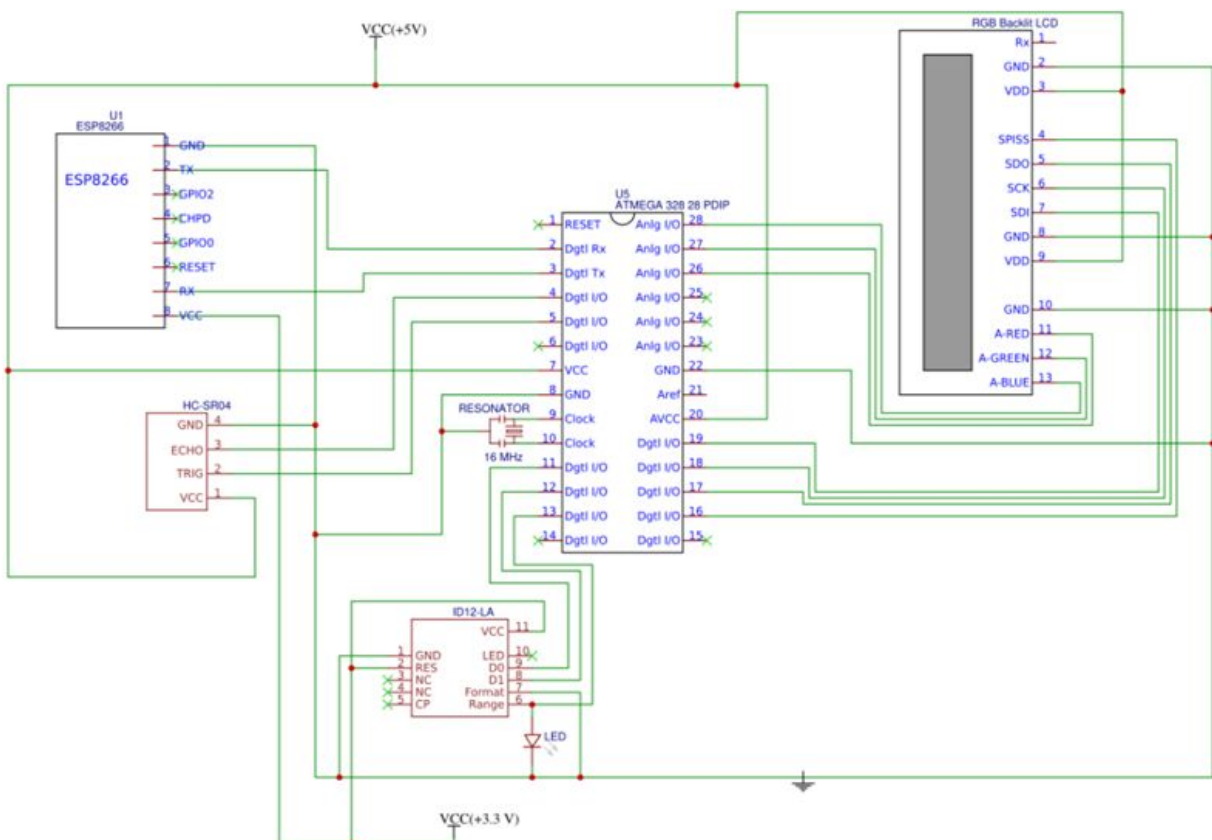


Figure 5: Circuit Diagram

Figure 5, above, shows the circuit diagram that shows how all the parts were connected. The ESP8266 WiFi Module(top left) can be seen to be relatively simple, needing only a Tx/Rx connection to the ATmega(center) for full functionality. In terms of circuit complexity, the LCD screen(top right) was the most difficult, considering the 3 voltage dividers required for its RGB backlight and the pins needed for I2C communication. The RFID chip(bottom), due to its built-in antenna, only needed power, ground, and a data pin to relay any scanned RFID tags. The Ultrasonic sensor(middle left) needed only two GPIOs to measure and read distances.

2.5 Microcontroller Code

The entire meter was controlled by two pieces of code: One that ran on the microcontroller, and one that ran on the WiFi module and communicated with the backend. Section 3 discusses the latter piece of code.

The microcontroller code was designed to have two constantly running functions:

- receiveFromESP
 - The backend, once it has received data, processes it and sends it back as LCD commands to have the screen reflect data accordingly(e.g. a customer has not verified their car in a spot. The backend processes that data and changes the meter's screen from green to red).
- sendToESP
 - This function first collects UltraSonic and RFID data, then sends that data to the WiFi Module to send to the backend. Since sending data to the backend was an expensive operation, we elected to only send data to the backend if something had changed. For example, if a car entered the spot/left the spot, or once an RFID tag was scanned. However, we would not update the backend if nothing had changed. This meant that we had to detect any possible changes as fast as possible, so that the backend would not miss any data.

“sendToESP” was the function responsible for actually collecting the Ultrasonic and RFID data. Unfortunately, the Ultrasonic sensor was not as reliable as we'd expected. Although on average, it returned the correct distance, it would occasionally return a minimum or maximum distance(10 or 400 cm). Because of this, we would take 10 measurements over 3 seconds, and only use that set of measurements if all 10 agreed on the status of the car(present in the spot/not present at the spot, represented as over/under 80 cm). Figure 6 (below) shows this algorithm in flowchart form.

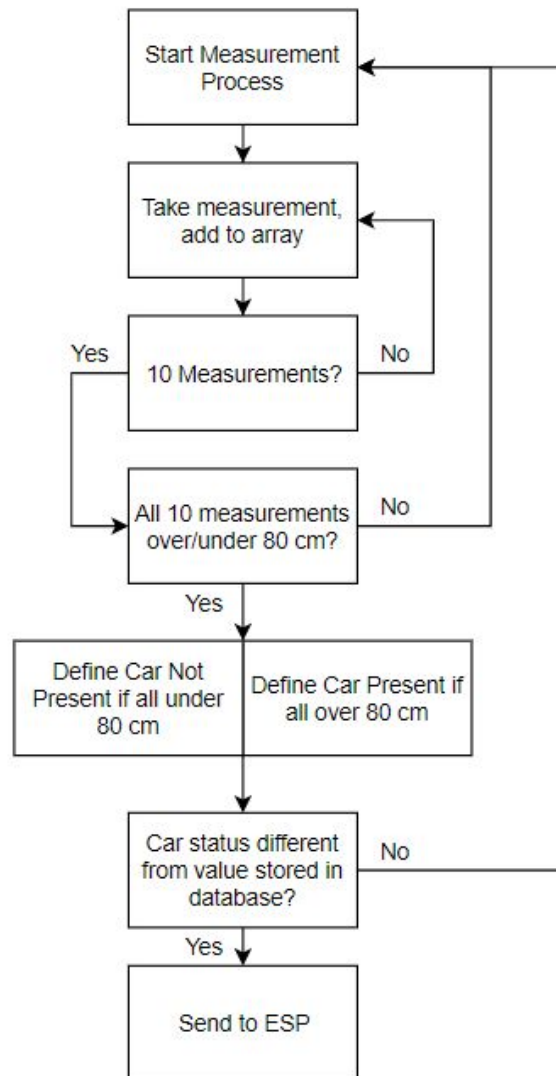


Figure 6: Ultrasonic Algorithm

3 Backend Design

3.1 Backend Overview

In the overall context of our system, the backend is essential for handling storage and most logic based operations. Specifically, the backend is the layer that operates between the app and the physical parking meter. There are a few different actions that trigger the backend logic: user interactions on the mobile app, users parking in a metered parking spot, and user scan of RFID tag. Additionally, there are functions that run on a time-based schedule. Appendix G provides a high level overview of the data communication between modules.

3.2 Firebase and Google Cloud

Firebase[9] and Google Cloud Platform[10] were what we chose as our backend technology stack. We chose Firebase because it links seamlessly with the Android mobile app that we designed for our project. Additionally, the FirebaseArduino library for the ESP8266 Wi-Fi chip was simple to use, and gave us the flexibility to use event listeners in our Wi-Fi chip code. We used the Firebase Realtime Database to store all of the data pertaining to our project. The database is stored in a NoSQL format, which means that the entire database can essentially be treated as a nested dictionary. We had three main tables in the database: Lots, Reservations, and Users. Each of the tables are structured as depicted below:

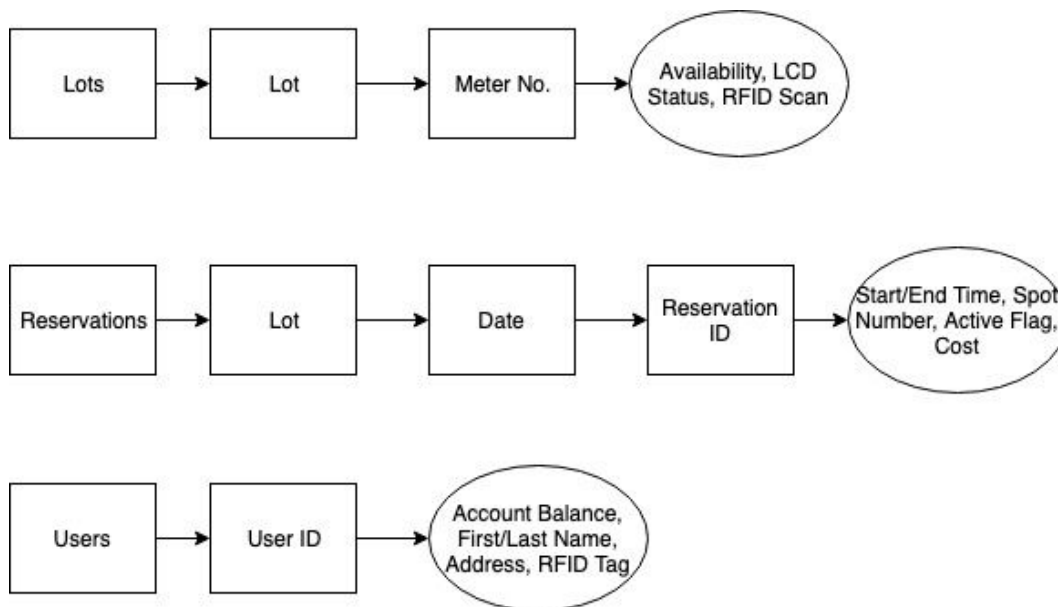


Figure 7: Database Table Structure

3.3 Wi-Fi Chip Code

Much like the microcontroller code that runs on the ATmega chip, the ESP8266 Wi-Fi chip has two constantly running functions. One function receives data through serial communication with the ATmega microcontroller and transmits this information to Firebase. The other function uses event listeners on certain Firebase paths to detect changes and send the changed information to the microcontroller. A more in-depth description of these two functions is detailed below.

- `sendToFirebase`
 - The `sendToFirebase` function is a constantly running function that has two main purposes. First, it receives data through serial communication from the microcontroller. Specifically, there are two different types of data that can be sent from the microcontroller to the Wi-Fi chip - ultrasonic sensor readings and RFID codes. Using specific start and end characters to indicate start and end of transmission [11], we were able to parse the data sent. Once the data is parsed, the appropriate Firebase datapath is updated using the `FirebaseArduino` library.
- `receiveFromFirebase`
 - The `receiveFromFirebase` function is another constantly running function that is needed to detect meter LCD color changes and update accordingly. To accomplish this, we used event listeners from the `FirebaseArduino` library on each meter's LCD color datapath. When these LCD color fields were changed by backend logic (reservation made, spot violation etc.), our event listeners would fire. At this point, we would pick up the new data from the database and send the updated color information to the microcontroller. Once again, we used start and end of transmission characters to communicate this data through serial.

3.3 Spot Assignment Algorithm

The spot assignment algorithm is a necessary element of our system, as it allows us to efficiently optimize the number of spaces that are reserved at any given point of time. During the creation of a reservation, a user only selects a particular lot. Users are not given the option to select a specific spot in this lot. Instead, a spot is assigned to the user at least an hour prior to the start of the reservation. For the purposes of our demo, we had the system assign a spot as soon as a reservation was created.

To assign spots, we deployed a Google Cloud Function[12] that is triggered by an HTTP POST request from the mobile app. That is, on submission of a reservation, the mobile app hits an endpoint called *makeReservation*. The POST request from the mobile app will carry a payload that contains a few

different fields necessary for the processing of a reservation. The contents of the payload include fields such as user ID, lot, date, start time and end time.

Once this POST request is received, there are a series of checks that are done to ensure that the reservation is valid. First, the cost of the reservation is computed and we check to see whether the specified user has enough account balance. If the user does not have sufficient funds, an error message is returned to the app. Next, the lot is checked for availability during the selected time period. If the lot is unavailable, an error message is returned to the app. If both of these checks pass, the system moves on to finding a spot for the reservation. The flowchart below describes how spots are assigned for a reservation.

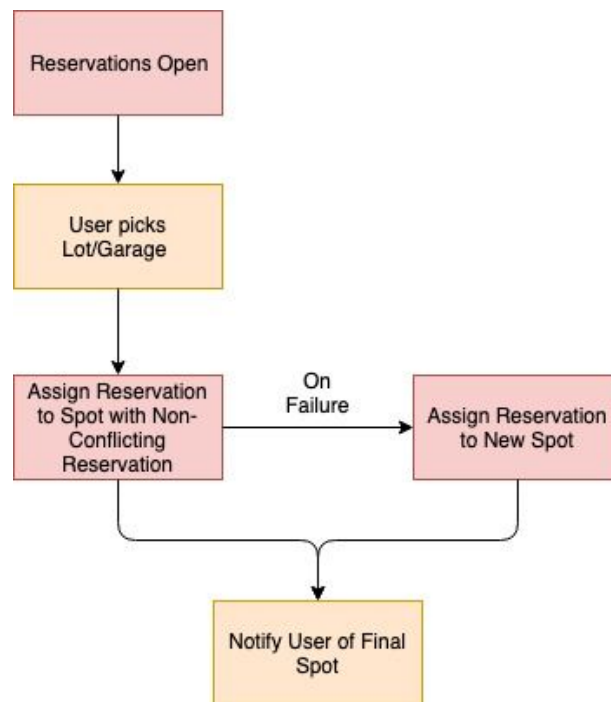


Figure 8: Spot Assignment Algorithm Flowchart

3.4 RFID Verification System

Another key component of our system was the RFID authentication module. In real world usage, each user would be mailed an RFID tag linked to their account. This RFID tag can be used to check in to a parking spot. With the RFID check in process, there are a few different scenarios that could play out.

- a) The user has a reservation, and is checking in to the reserved spot.
- b) The user does not have a reservation, and is parking in an open spot.
- c) The user is parking in a reserved spot, but does not have a reservation for that spot.
- d) The user never scans in to the spot (reserved or open)

To handle these different cases, we implemented a system that would correlate arrival time into the parking spot with RFID readings. Once a car has parked in a spot, the ultrasonic sensor picks up this change and changes spot status from open to closed. An event listener on this data path in Firebase then picks this change up and triggers the RFID verification system. For purposes of a shorter demo, we gave users 30 seconds to scan into the meter before the spot is marked as being in violation. The flowchart below illustrates how this system works.

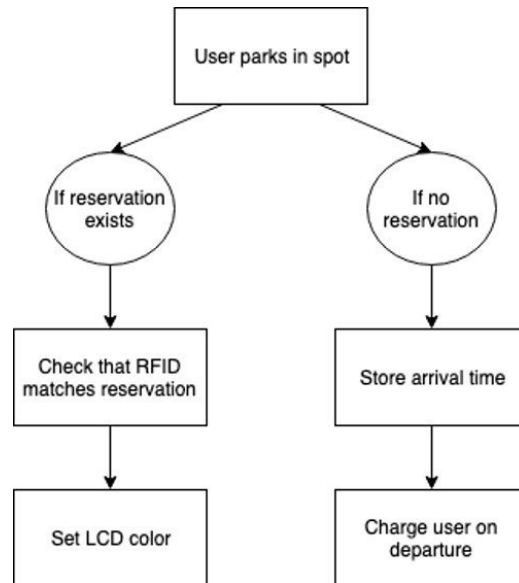


Figure 9: RFID Verification Flowchart

3.5 Reservation Enforcement System

The final piece of our backend was a reservation enforcement system. The reservation enforcement system runs on a time-based schedule, and is currently set to execute every minute.

While fairly simple, this system was required to cover two edge cases:

- Reservation overstay - When a user stays in a reserved spot past their reservation end time, the enforcement system automatically changes the status of the specified spot to being in violation.
- Reservation no-show - If a user does not check-in to their reserved spot within 30 minutes of their start time, the system deletes the reservation from the system and frees up the spot for parking.

4 Mobile App Design

4.1 Mobile App Overview

The mobile app was a central component to our project. The app allows users of the Parking Reservation System to sign up for the service, update user information, make/view reservations, and view parking spot availability. The app is designed for Android devices[5] and is written in Java. We chose to design the app for Android devices because Android applications integrate well with our backend service, Firebase.

4.2 Mobile App Frontend Design

There are four pages in the mobile app: Login, User Profile, Reservation, Open Spots. The user can interact with the app as depicted below:

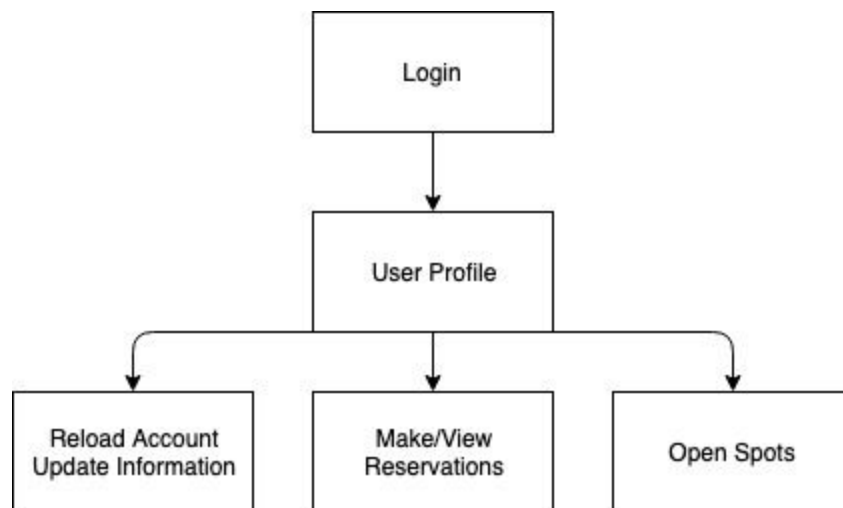


Figure 10: App Interaction Flowchart

The Login page has two functions, it allows users to login and it allows users to create an account. The User Profile page contains all of the user's personal information such as their name and address. In addition, this page allows users to add more balance to their account and sign out of the app to end their session. The Reservation page allows users to make and view reservations. Users can make reservations by selecting a lot and specifying a start and end time using the time pickers that appear upon clicking the respective fields on the page. When a user submits their reservation request, they receive feedback saying: "Reservation Successful", "No Spots Available", or "Insufficient Balance". If the reservation is successful, it appears on the table in the bottom of the page along with the spot number

once it has been assigned by the system. The Open Spots page allows users to view spot availability at the lot of their choice. Once the user has selected the lot, they can view that status of each meter whether it is “OPEN”, “RESERVED”, or “OCCUPIED.”

Images of all of the pages are included in Appendix E.

4.3 Mobile App Backend Design

Some of the backend processing is handled on the app itself. The app handles all of the processing when a user creates an account. Once an account is created, the app passes the information on to Firebase where the userid and password are stored in the user database until the account is deleted. In addition, the app verifies if a given user ID and password pair match their respective pair in the database and allows the user to login. If a user updates any of their personal information or reloads their account balance on the User Profile page the change is reflected immediately in the database.

On the Open Spots page, the app reads the color value for each meter from the database and displays the appropriate text whether it is “OPEN”, “RESERVED”, or “OCCUPIED.” On the Make Reservation page, the app makes a POST request to the Google Cloud Function upon the user submitting their reservation request and displays the response received from the function to the user. In addition the app reads the database for reservations corresponding to the user and displays them row by row in a table. The app also displays the spot number in the table once it has been assigned by the Google Cloud Function and updated in the database.

5 Verification

During the build phase of the project, each component was tested to ensure proper and full functionality. The components were all integrated with no (unsolvable) issues, and the final product now functions properly. All of the verification tests can be seen below in Appendix A: Requirements and Verification Table.

5.1 Hardware Verification

The entire circuit was built out and rigorously tested on a breadboard prior to even designing the PCB. Due to that, the PCB was relatively bug-free after it was actually printed. With the exception of routing one trace on the front PCB (see Appendix D: PCB Boards), the PCB worked perfectly and did not need any changes.

5.2 Backend Verification

The backend of our system is fairly complex and has many moving pieces that need to work together in unison for everything to work as expected. Rigorous verification of each individual piece of code was absolutely necessary for debugging.

The first module that was tested was the Wi-Fi module. There were two main tests that were run on the Wi-Fi chip. First, we checked to see whether we could receive basic text through serial. Once we were able to confirm that the microcontroller to Wi-Fi chip code was fully functional, we then moved over to the Firebase communication side. On the Firebase end, we verified that we could push data to a specified datapath and also checked that our event listeners fired on modification of data values.

To test that our Google Cloud Functions were working, we first tested that the logic worked locally. To accomplish this, we listed all possible edge cases and ran through expected functionality versus actual functionality. Once this code was deployed, most of our code worked right out of the box. There were a few issues with regards to server timezone, but that was easily fixed.

5.3 Mobile App Verification

The mobile app verification tests consisted of performing actions on the app and seeing if the database and meter responded appropriately. A series of simple queries [6] were ran to see if new data created by the app was present in the database and to see if any updates to user data on the app was also reflected in the database. In addition, we ran multiple tests to see if reservations made on the app reserved the appropriate meter. We also ran multiple tests to see if the app was accurately displaying the current meter status, occupied or available.

6 Costs

6.1 Labor

Person	Hours	Hourly Rate	Total
Manjesh Mogallapalli	30	\$50	\$1500
Ojus Deshmukh	30	\$50	\$1500
Vivek Calambur	30	\$50	\$1500

6.2 Parts

Part	Qty	Total Cost(\$)
ESP8266-01 Wifi Module	4	\$13.98
ATMega 328-P	4	\$12.99
ID-12LA RFID Reader	3	\$89.85
32-Bit RFID Tags	6	\$23.70
HC-SR04 Ultrasonic Sensor	3	\$11.85
LCD Screen	3	\$62.64
680 Ω Resistor	3	\$1.00
1000 Ω Resistor	3	\$1.00
Ceramic Resonator	3	\$2.85
3.3V Regulator	3	\$7.48
5V AC/DC Converter	3	\$29.97
Total		\$257.31

7 Conclusion

7.1 Accomplishments

We were able to accomplish everything that we had planned for over the course of the semester. We met all of the high-level requirements that we set in the Design Review, and we designed a completely functional product that solves a relevant problem.

Our Parking Reservation System allows users to view parking spot status in realtime, it allows users to make reservations using the mobile app, the system takes care of verifying reservations via RFID tag and backend processing, and the system accurately enforces reservations and payment without any hassle to the user.

7.2 Challenges

During our time working on this project we encountered a few major challenges. One of the challenges we faced was a bug regarding the ultrasonic sensor. The ultrasonic sensor would time-out if no new activity happened after a couple of minutes, and it would retain whatever the last value it had. We believe this happened due to the quality of the ultrasonic sensor, as it is very inexpensive, a lot of other users were complaining about experiencing a similar issue on online forums.

We also experienced an issue with the FirebaseArduino library that we used on our WiFi chip. On the day before the demo, we were experiencing issues connecting our WiFi chip to the database, and we couldn't figure out why as all of our code was working before. After looking at various online forums, we found that the issue was the FirebaseArduino library that we were using refers to the Firebase website certificate and whenever the library gets updated the certificate changes. So once we changed the certificate key to the updated one, we were once again able to connect to the database.

Another challenge that we faced was listening for changes that occurred in the database, "events." In Firebase, PUT and PATCH are two events that write to the database. PUT writes a new entry to the database whereas PATCH overwrites an existing entry with updated information. Initially we were only checking for PUT events, but once we checked for PATCH events, which happened any time we wanted to update the LCD screen based on the backend logic, we encountered bugs in the FirebaseArduino library where the information associated with a PATCH event was empty. In order to work around this bug, we had to write a workaround that retrieved the LCD color value from the DB every time we received a PATCH event. This color code was then sent to the microcontroller to update the screen color.

We also experienced problems with getting our code for the Wifi chip to stay on the chip once we disconnected it from the breadboard. We realized to program the wifi chip GPIO pin must be set to

low and for the program to stay on the chip the GPIO pin must be set to high. This discovery was only made after looking through some online forums, as the documentation didn't specify anything about this.

A big challenge for our group was building the mobile application. While all of us have some sort of software experience, none of us have ever worked on mobile development. So we had to learn how to build a mobile app from scratch, which required reading a lot of documentation and online forums.

7.3 Future Work

There are a few improvements that we would like to make to our project in the future. We would like to look into pivoting from the current ultrasonic sensor as it has the issues that we described above. We would either move on to a higher quality ultrasonic sensor or look into using an infrared sensor.

In addition, there were a few modifications we would like to make to the software component of our project. We would like to look into transitioning from Firebase to a commercial backend service such as AWS. While Firebase served as the perfect backend service for what we were trying to accomplish for this course, AWS is considered to be the backend service of choice in the tech industry. We also want to develop a mobile app for Apple devices as it would allow our product to be used by many more users.

Finally, we also want to modify our physical design to make it a completely waterproof product. Our current design has a wire routed through the outside of the pole to the ultrasonic sensor, and we would like to change that so all of the wires are contained within the casing for the meter.

7.4 Ethical Considerations

There were initially two overarching ethical considerations that we discussed in our design document - physical safety and data privacy. In terms of physical safety, our project addresses the concern of mobile phone usage when driving. We assign spots well ahead of user arrival giving users no extra incentive to use their mobile phones while driving. In terms of data privacy, all private user information is encrypted and stored on Firebase. Additionally, Firebase provides built-in security features that restrict access to the database without an access key. We believe that our project is well aligned with the IEEE Code of Ethics[7] as well as the ACM Code of Ethics[8].

7.5 Acknowledgements

We would like to thank TAs Kyle Michal and Soumithri Bala for all of their assistance throughout our time working on this project.

8 References

- [1] P. Sawyer, “Motorists spend four days a year looking for a parking space,” *The Telegraph*, 01-Feb-2017. [Online]. Available: <http://www.telegraph.co.uk/news/2017/02/01/motorists-spend-four-days-year-looking-parking-space/>. [Accessed: 02-May-2019].
- [2] “Searching for Parking Costs Americans \$73 Billion ... - INRIX.” [Online]. Available: <http://inrix.com/press-releases/parking-pain-us/>. [Accessed: 01-May-2019].
- [3] FirebaseExtended, “FirebaseExtended/firebase-arduino,” *GitHub*, 19-Apr-2019. [Online]. Available: <https://github.com/FirebaseExtended/firebase-arduino>. [Accessed: 02-May-2019].
- [4] “Wire,” *Arduino*. [Online]. Available: <https://www.arduino.cc/en/Reference/Wire>. [Accessed: 02-May-2019].
- [5] “Meet Android Studio | Android Developers,” *Android Developers*. [Online]. Available: <https://developer.android.com/studio/intro>. [Accessed: 02-May-2019].
- [6] “Query | JavaScript SDK | Firebase,” *Google*. [Online]. Available: <https://firebase.google.com/docs/reference/js/firebase.database.Query>. [Accessed: 02-May-2019].
- [7] “IEEE Code of Ethics,” *IEEE*. [Online]. Available: <http://www.ieee.org/about/corporate/governance/p7-8.html>. [Accessed: 02-May-2019].
- [8] “Code of Ethics,” *ACM Ethics*, 29-Jul-2016. [Online]. Available: <https://ethics.acm.org/code-of-ethics/>. [Accessed: 02-May-2019].
- [9] “Firebase,” *Google*. [Online]. Available: <https://firebase.google.com/>. [Accessed: 02-May-2019].
- [10] “Google Cloud Platform,” *Google*. [Online]. Available: <https://cloud.google.com/>. [Accessed: 02-May-2019].
- [11] “Cloud Functions - Event-driven Serverless Computing | Cloud Functions | Google Cloud,” *Google*. [Online]. Available: <https://cloud.google.com/functions>. [Accessed: 02-May-2019].
- [12] *Serial Input Basics - updated*. [Online]. Available: <https://forum.arduino.cc/index.php?topic=396450.0>. [Accessed: 02-May-2019].

Appendix A Requirements and Verification Table

Component	Requirement	Verification
Power Supply Subsystem	The power subsystem should be able to fully power all components on the meter, at all times.	✓ Check if the meter can be powered by the Power Supply subsystem, and carry out all tasks required(retrieve sensor data, communicate using WiFi module
3.3V Regulator	The 3.3V regulator should be very close (+/- 5%) to 3.3V to avoid burning out the WiFi module	✓ Test the 3.3V regulator with a multimeter to see if it regularly reads within 5% of 3.3V
Ceramic Resonator	The resonator should allow the ATmega to function at 16 MHz.	✓ Measure the frequency of the two legs of the resonator with a multimeter to check the frequency
WiFi Module	The WiFi module should be able to connect to the backend.	✓ Have the WiFi module randomly generate a number between 1 and 5. Have the onboard LED blink that many times, then upload that same number to a table in Firebase. Check to see that the two values are the same.
WiFi Module	The WiFi module should be able to pass commands to the meter	✓ Pass commands from Firebase through the WiFi module to change the color/text on the LCD
WiFi Module	The WiFi module should be able to receive sensor data from the meter	✓ Print Ultrasonic measurements on the serial monitor, while also passing them through the WiFi module to Firebase. Check to see that the two values are the same.
RFID Module	The RFID reader should be able to read RFID tags and pass their RFID codes over serial.	✓ Use serial monitor to read and print RFID codes scanned from tags

Ultrasonic Sensor	The Ultrasonic Sensor should be able to accurately measure distances(+/- 15 cm)	✓ Have the Ultrasonic measurements print over the Serial monitor. Using measuring tape, place a box at different distances from the sensor. Confirm printed and placement distances are the same
LCD Screen	The LCD screen should be able to be controlled over I2C and change its screen color.	✓ Print multiple messages and colors on the LCD screen.
RGB Backlight	The RGB Backlight needs 3 voltage dividers supplying 1.9,2.9,and 2.9V, respectively.	✓ Use a multimeter to confirm the voltage divider is built properly.
Mobile App	Users must be able to create an account	✓ Check database with a simple query to see if new user is present.
Mobile App	Users must be able to login to account with their credentials	✓ Check mobile application to see if it allows users to login with correct credentials. ✓ Check mobile application to see if it allows users to login with incorrect credentials.
Mobile App	Mobile application will display open parking spots	✓ Cross check open spots on mobile application with those on database to ensure the same spots are being shown as open
Mobile App	User is able to edit personal information	✓ Check database with simple query to see if new changes are reflected for the specific user
Mobile App	User is able to edit account balance	✓ Check database with simple query to see if account balance is present in the database
Mobile App	User can make a reservation.	✓ Check database with simple query to see if reservation is present in the database
Backend	Reservations are stored in database	✓ Use Postman to send a POST request and see if reservation is created in Firebase

Backend	RFID scan is reflected in database	✓ Scan an RFID tag and ensure that the correct RFID code shows up in the database
Backend	User is charged appropriately when user parks in open spot	✓ Check account balance of specified user and ensure that the correct amount was deducted
Backend	User is authenticated correctly when user has an active reservation for the spot	✓ Ensure that screen turns off when user parks in spot corresponding to reservation
Backend	Spot is in violation when unauthorized user parks in reserved spot	✓ Scan RFID tag of user who does not have an active reservation and check if LCD screen turns red
Backend	Spot is in violation when user overstays the reservation	✓ Ensure that screen turns red when user stays in spot after reservation end time

Appendix B PCB Schematics

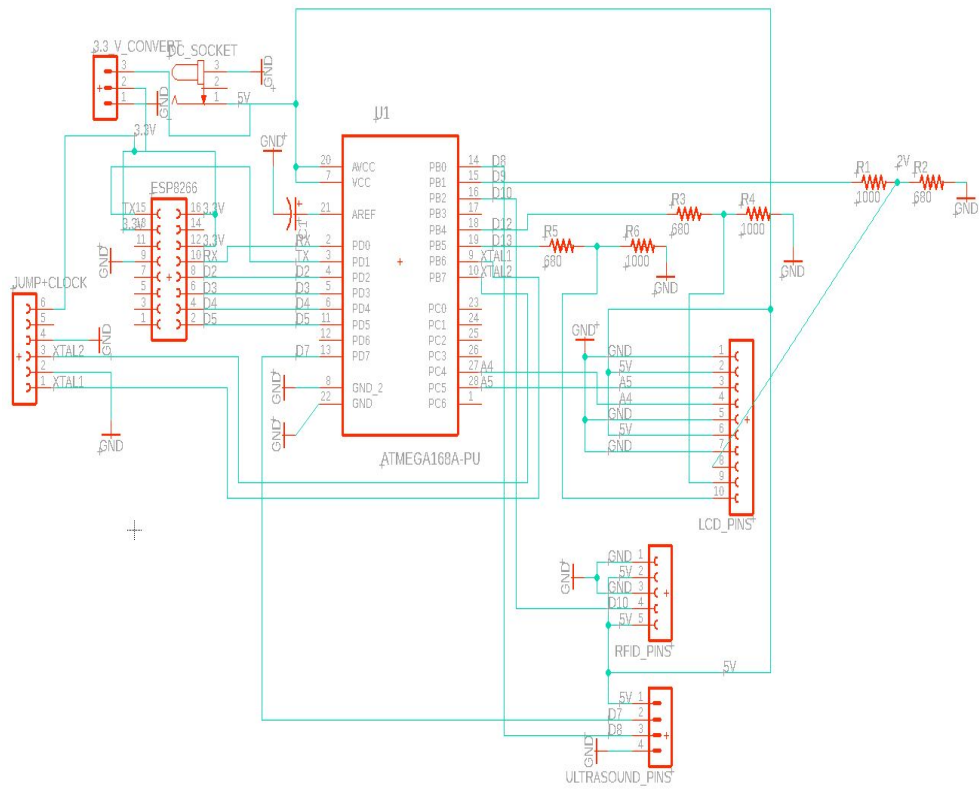


Figure 11: Main PCB Schematic

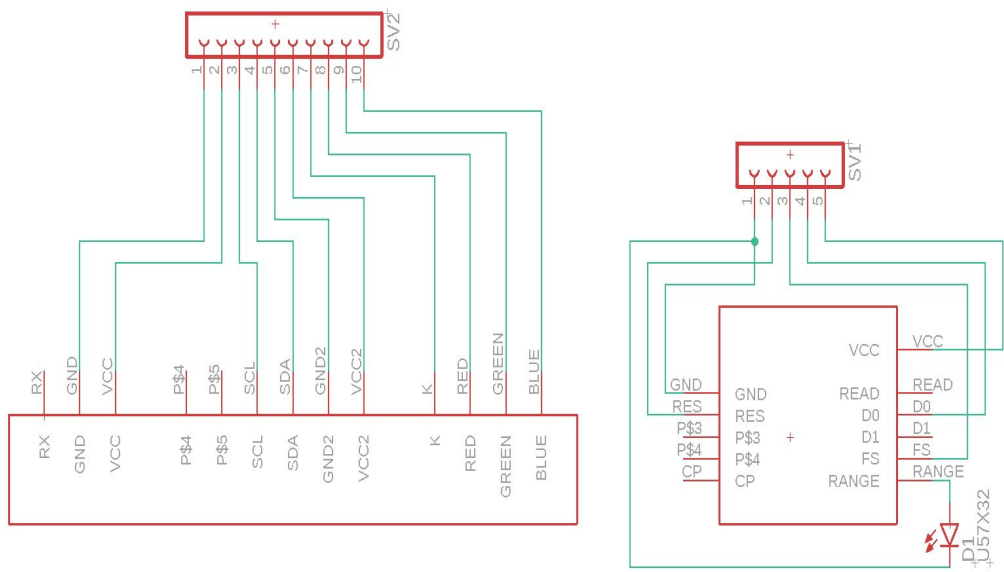


Figure 12: Front PCB Schematic

Appendix C PCB Layouts

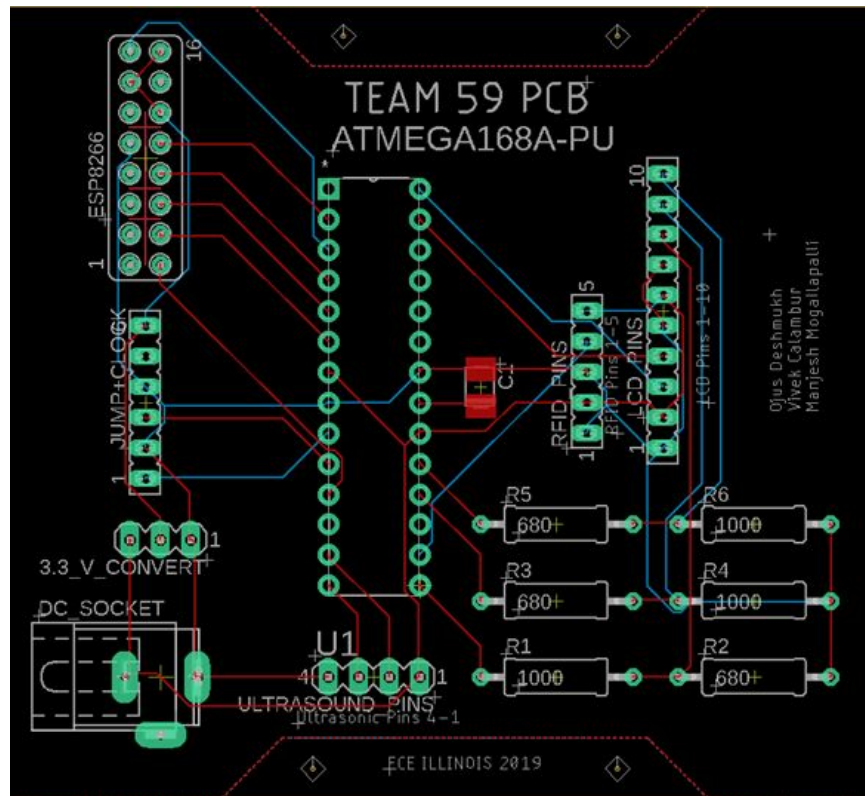


Figure 13: Main PCB Layout

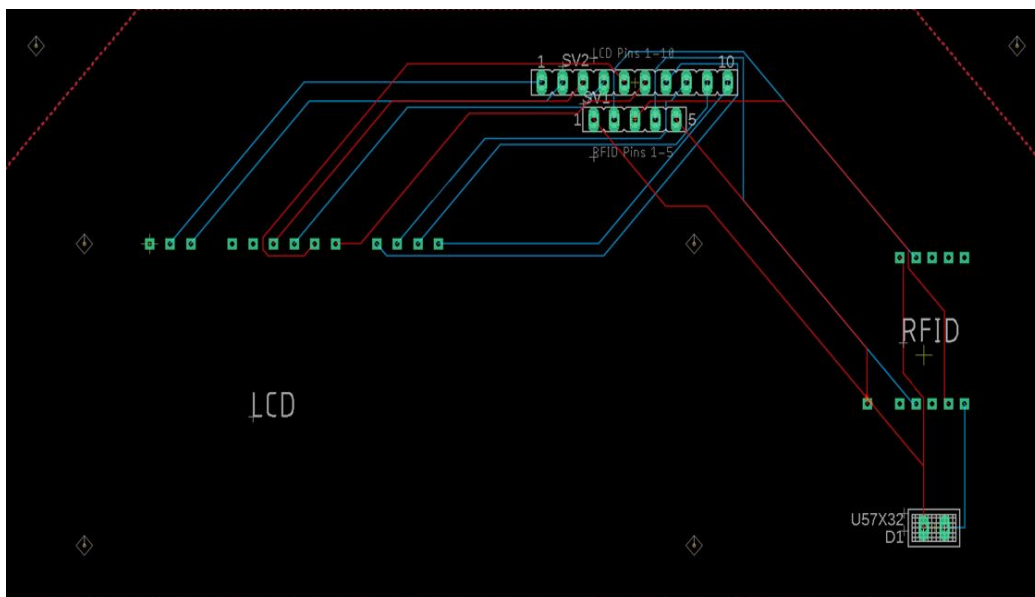


Figure 14: Front PCB Layout

Appendix D PCB Pictures

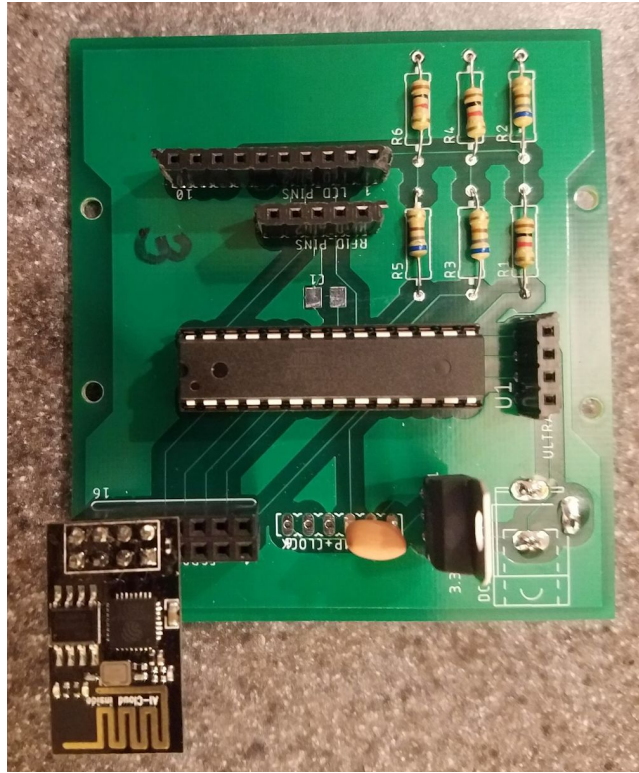


Figure 15: Main PCB Board

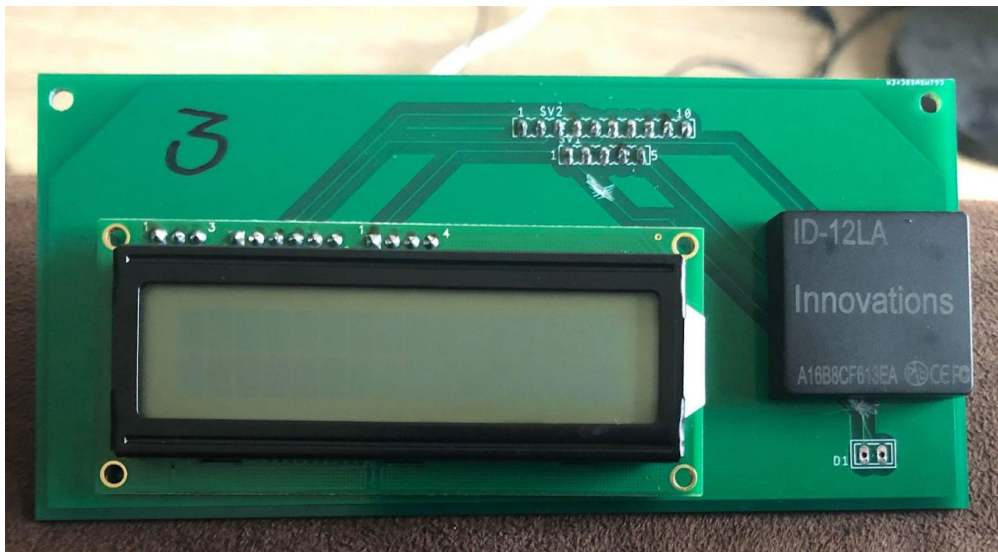


Figure 16: Front PCB Board

Appendix E Physical Design

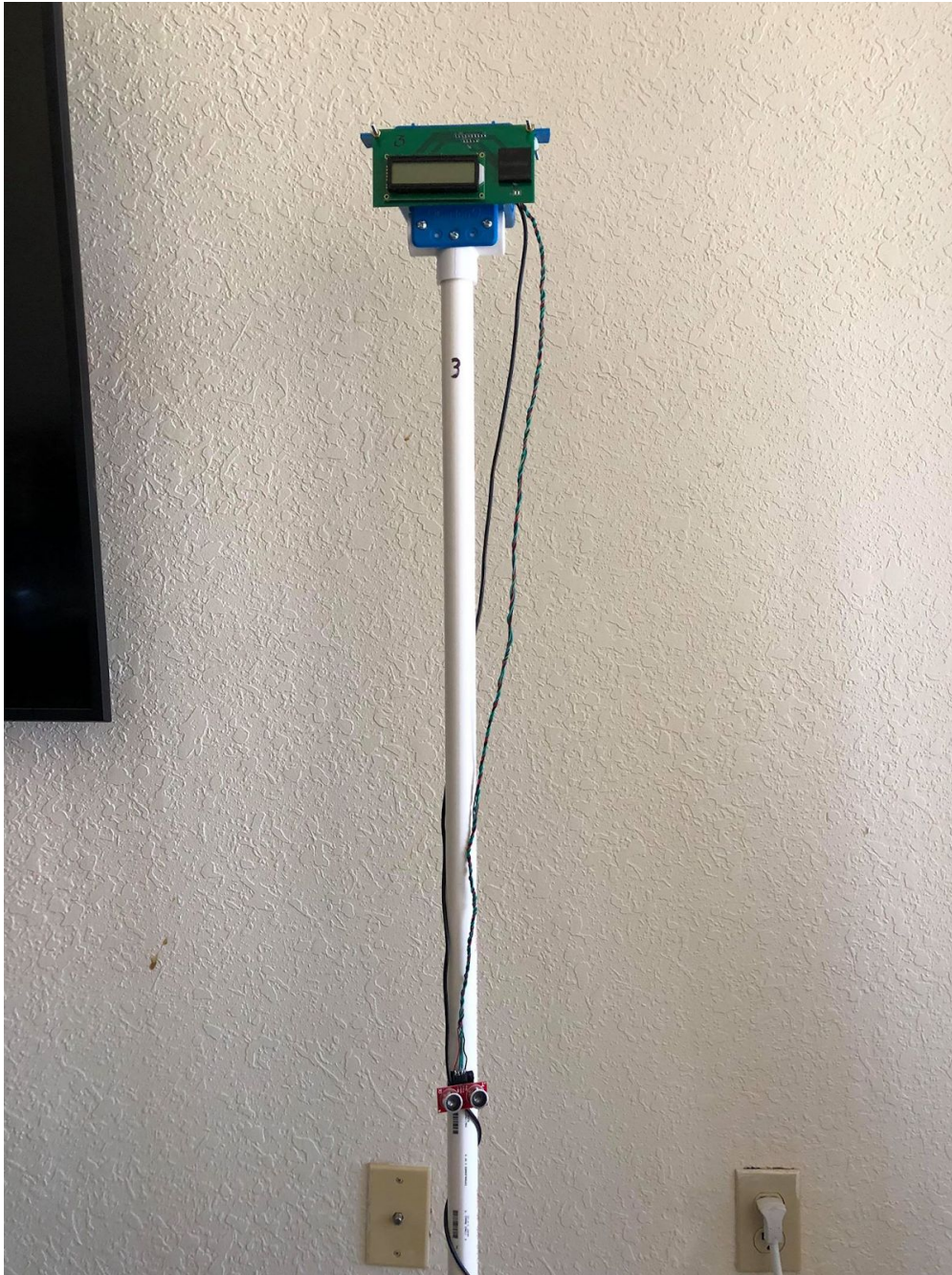


Figure 17: Full Physical Design



Figure 18: Zoomed-in Physical Design

Appendix F Mobile App Layout

Parking Reservation App

Welcome to the Parking Reservation App


Username

Password

SIGN IN

CREATE ACCOUNT

UserProfile



User Profile

Manjesh

Mogallapalli

508 East Healey St Apt. 220

Champaign

Illinois

61821

UPDATE PROFILE

Account Balance

103

RELOAD \$5

SIGN OUT

Reservations

Open Spots

Profile

Figure 19: Login and user profile pages

MakeReservation

Make a Reservation

Reservations are for 26-Apr-2019

Lot: ECEB

\$3 per Hour

Start Time

End Time

SUBMIT

Current Reservations

Lot Name	Start Time	End Time	Spot Number
----------	------------	----------	-------------

Reservations

Open Spots

Profile

OpenSpots

Open Spots

Lot: ECEB

Meter #1

OPEN

Meter #2

OPEN

Meter #3

OPEN

Reservations

Open Spots

Profile

Figure 20: Reservations and open spots pages

Appendix G Schedule

Week	Manjesh	Vivek	Ojus
2/25/19	Prepare for Design Review.	Prepare for Design Review.	Prepare for Design Review.
3/4/19	Begin the framework for the mobile application.	Set up AWS environment and write the framework for DB connection	Begin testing and verification for all components.
3/11/19	Finish writing mobile application. Help Ojus with PCB Design	Begin writing firmware code for data transmission to/from PCB.	Complete verification of all components. Design and order PCB.
3/18/19 (Spring Break)	Perform initial testing on mobile application. Comment and clean up existing code.	Perform initial testing on firmware and backend connection. Comment and clean up existing code.	Collaborate with others to ensure software is ready for integration.
3/25/19	Work with Vivek on assembling physical and electrical components. Help Ojus with PCB testing.	Assemble all physical components and electrical components.	Ensure ordered PCBs meet functionality requirements and order additional PCB if necessary.
4/1/19	Help Ojus with physical and electrical component integration. Integrate software with all hardware components.	Complete the final firmware code for the final PCB design.	Complete physical and electrical components integration.
4/8/19	Begin final testing of the complete system. Focus on issues regarding mobile application.	Begin final testing of the complete system. Focus on issues regarding Firmware.	Begin final testing of the complete system. Focus on issues regarding PCB performance.
4/15/19	Fix all mobile application issues. Ensure mobile application meets requirements.	Fix all Firmware issues. Ensure that all firmware meets requirements.	Fix all PCB issues. Ensure that all physical components of project are functional.

4/22/19	Ensure entire parking reservation system is complete and begin Final Report.	Ensure entire parking reservation system is complete and begin Final Report.	Ensure entire parking reservation system is complete and begin Final Report.
4/29/19	Final Report & Presentation	Final Report & Presentation	Final Report & Presentation

Appendix H Project Data Flow

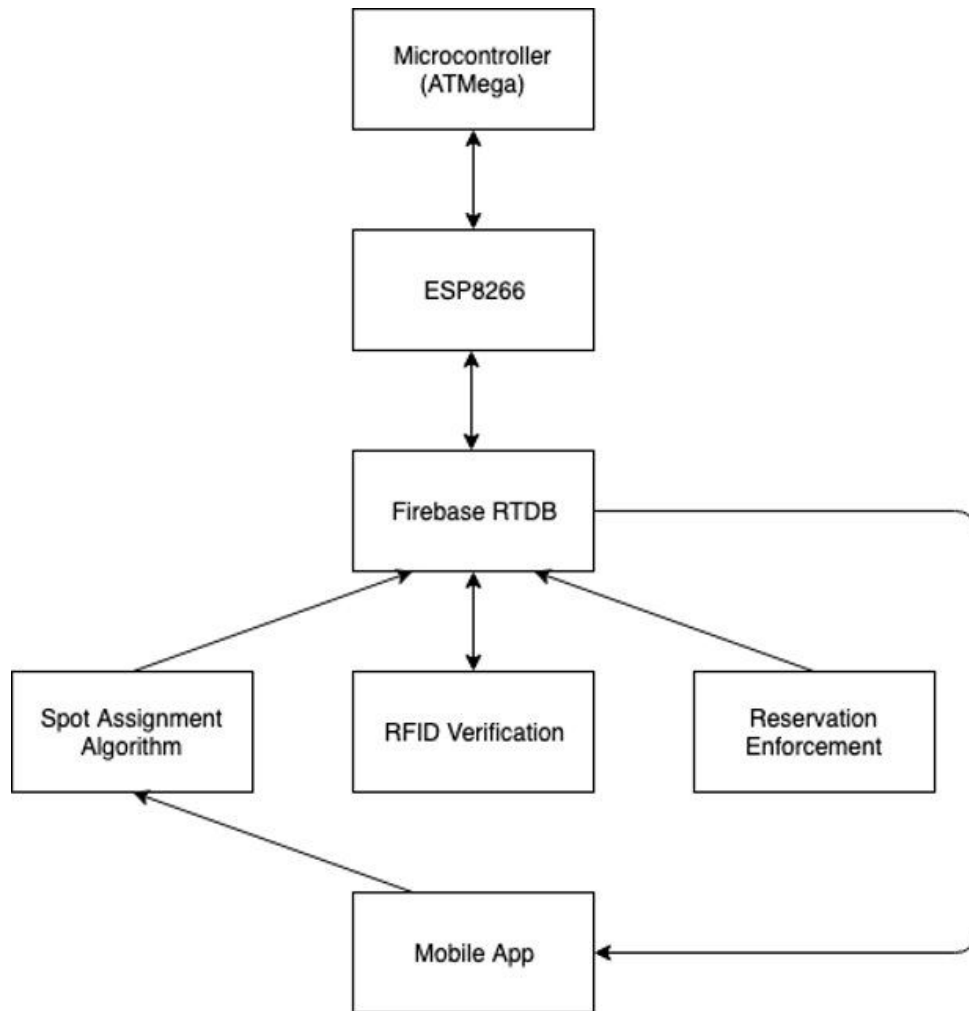


Figure 23: Complete Project Data Flow